Team 4: Rong Cui, Xuanxuan Ge, Ben Lin, Xinyi Tao
rongcui, xuanxuag, cblin, xinyit

17480/17780 Final Report

1. API Fixed and Scope

In this project, we fixed part of the Java Gitlab API, a wrapper written in Java of the Gitlab API. For the sake of time, we limited our scope to the most frequently used gitlab components including GitlabUser, GitlabProject, GitlabSession, GitlabBranch, GitlabCommit, GitlabIssue, and GitlabMergeRequest, from the viewpoint of a user with normal permission. Documentation of our fixed API is available in the form of javadoc.

2. Problems of the API

Please refer to section 1 in our project proposal.

3. Our fix

The high-level rectifications we made can be summarized as the following:

- Organize all gitlab components into a hierarchical structure where the operations of each component is managed in the parent component.
- Merge the function of GitlabSession into the root class `GitlabAPIClient` because with the hierarchical structure discussed above, the instance extracted from the parent instance should be valid as long as the parent instance has valid authentication.
- Use Builder Pattern for flexible construction and edit before CRUD operations

The rest of this section focuses on the comparison between our version of the API with the original Java GitlabAPI. More details about design decisions made through the multiple iterations we did are discussed in the next section.

3.1 APIClient

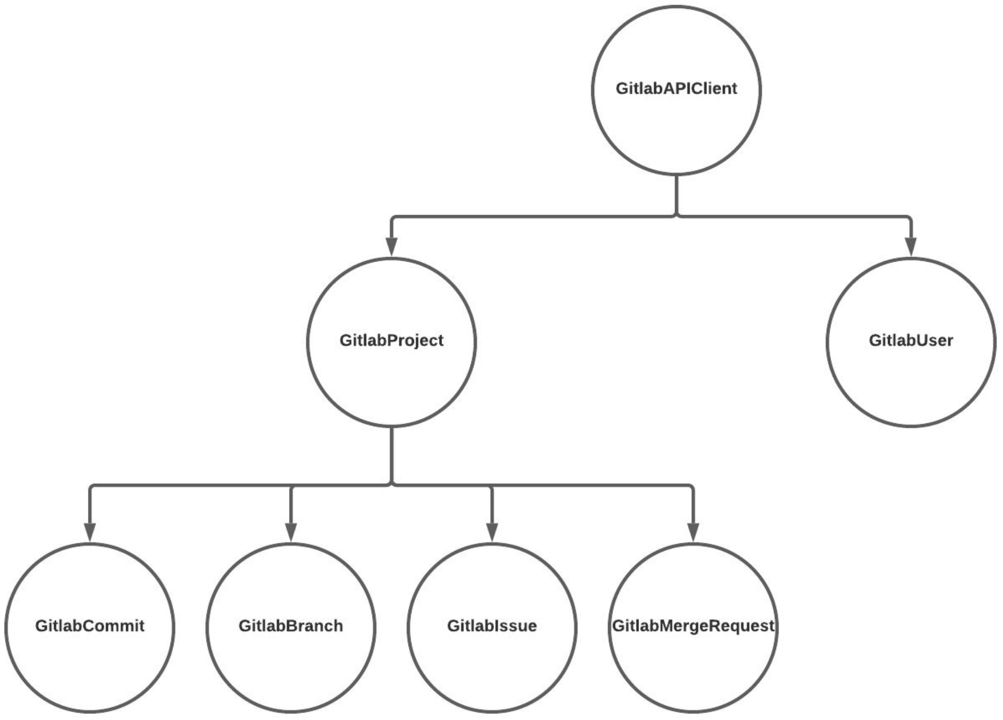
The original API has 5 overloaded connect() methods, which are confusing and error prone because they violate consistent parameters ordering. It's very easy to misuse the creation method for GitlabAPI instances. Instead, the fixed version of APIClient is named GitlabAPIClient, and we utilized the builder pattern with only the endpoint as the required field. This builder handles a bunch of optional parameters and enables flexible and less error-prone construction of API clients. Furthermore, we support new features like setting HTTP request configurations in this builder with internally embedded HTTP client helper.

```
// Original API: obtain API handle with access token and namespace
String hostUrl = "gitlab.com";
String apiToken = "token";
String apiNameSpace="api/v4"
GitlabAPI api = GitlabAPI.connect(hostUrl, apiToken, TokenType.ACCESS_TOKEN,
apiNameSpace);
// Fixed API: obtain client with access token and namespace
GitlabAPIClient client = new GitlabAPIClient
                           .Builder("gitlab.com")
                           .withAccessToken("token")
                           .withApiNamespace("api/v4")
                           .build();
```

3.2 Hierarchical Structure

The original GitlabAPI has a flat structure and exposes a GitlabAPI class for users to access all of the components they support. GitlabAPI class handles creating, querying, updating and deleting all of the components they support. It’s hard to read and learn this whole API since there are over 300 public methods in total. Also, the JavaDoc for the original API is insufficient and thus the readability of this API can be very poor.

Figure below shows how our fix organizes all of the components in a hierarchical structure. Creation of GitlabProject is done using calling newProject() method inside GitlabAPIClient. Similarly, creation of GitlabBranch is done using the newBranch() method inside GitlabProject.



3.3 CRUD

The original GitlabAPI has an extremely long parameter list when it comes to creating/updating a component. Below is a comparison of creating a project and updating the name of the project.

```java
// Original API:
GitlabProject project = api.createProject("name", namespaceId, true, false,
                        true, false, true, true, visibility, "importUrl");
GitlabProject updatedProject = api.updateProject(project.getId(), "newName",
                               namespaceId, true, false, true, false, true,
                               true, visibility, "importUrl");
List<GitlabProject> projects = api.searchProjects("foo");
api.deleteProject(project.getId());


// Fixed API:
GitlabProject project = client.newProject("name")
                .withDescription("This is Description")
                .withIssuesEnabled(true)
                .withJobsEnabled(false)
                .withWikiEnabled(true)
                .withVisibility(visibility)
                .create();
GitlabProject updatedProject = project.withName("newName").update();
List<GitlabProject> projects = client.getProjectsQuery()
                                     .withSearch("foo")
                                     .query();
updatedProject.delete();
```

3.4 Error Handling

The original Gitlab API throws IOExceptions if any HTTP request such as create, read, update, and delete fails. We created our own uncheck GitlabException class which handles all of the IOException which might occur during HTTP request, and we will add meaningful messages taken from Gitlab Web API to let users know what went wrong during the API call.

3.5 Missing Functionality Supported

We have added missing getters as described in section 2 of the report to support common user requirements. For example, users are able to get a closedAt date from GitlabIssue component whereas the original API failed to provide all the getter that the Gitlab Web API provides.

4. Major Design Decisions

Since this is a general fix for original Java-Gitlab-API, we mainly focus on resolving existing problems and proposing some new features to improve its functionality, usability, quality, and readability.

4.1 API Hierarchy & Functionality Partitioning

The biggest problem in Java-Gitlab-API is that all utility methods exist in the same class GitlabAPI, which results in a gigantic class. Thus, we decided to partition the functionality into smaller parts and improve the usability and functionality of this API. Basically, we should support create/read/update/delete (CRUD) for Gitlab components to specified endpoints with pre-declared credentials.

We made an abstraction of "session" to represent and maintain the authentication and connection configurations. In this way, we can keep the credential (probably a token) and the endpoint in an encapsulated and immutable object, and maintain it in every component instance. Through this, users can use the same set of configurations to invoke Gitlab REST API calls and that's what's preferable to do. Even though REST API is stateless, we keep the virtual "session" information in the background.

To reduce the number of methods in the main class, **we maintained a hierarchy of component classes**. (As shown in Section 3.2) User requests should start from GitlabAPIClient where we can get GitlabProject and GitlabUser. This structure keeps consistent behaviors among components and makes it appropriate for Gitlab API users instead of only for Java-Gitlab-API users. Note that the API's audience should be any Gitlab API user but not some specialist for this wrapper API.

```
// Say if we want to access a branch's protected status of a given project.

// The original API: (hard to use)
String hostUrl = "gitlab.com";
String apiToken = "token";
String apiNameSpace="api/v4"
GitlabAPI api = GitlabAPI.connect(hostUrl, apiToken, TokenType.ACCESS_TOKEN,
apiNameSpace);
GitlabProject project = api.getProject(projectId);
GitlabBranch branch = api.getBranch(project, branchName);
boolean isProtected = branch.isProtected();

// The fixed API: (easy to use and comprehensive workflow)
GitlabAPIClient client = new GitlabAPIClient
                          .Builder("gitlab.com")
                          .withAccessToken("token")
                          .withApiNamespace("api/v4")
                          .build();
GitlabProject project = client.getProject(projectId);
GitlabBranch branch = project.getBranch(branchName);
boolean isProtected = branch.isProtected();
```

Having the hierarchies among components, another decisive question can be how and where to operate on Gitlab components. To make it easy to do what's preferable, **we decided to only expose any existing component from its parent component** and **builder methods from who a component to be created belongs to**. CRUD operations were enabled in each component class to let users operate on the instances directly. Note that some component classes may not support create or update methods since they are not supported by Gitlab API or not allowed for users without admin access.

There are two alternatives of where the operations take place: one is to perform CRUD operation to one component where it belongs to, another is to let each component hold everything it requires and perform CRUD operation by itself.

```
// Say if we can GitlabProject instance `project` and want to create an issue named
`foo` and then update its description as `desc` and then delete it.

// First choice is to create issue first and invoke create issue from project
GitlabIssue issueFoo = GitlabIssue.Builder("foo").build();
project.createIssue(issueFoo);
issueFoo.setDescription("desc");
project.updateIssue(issueFoo);
project.deleteIssue(issueFoo);

// Second choice is to build an issue from the project and call create freely.
GitlabIssue issueFoo = project.newIssue("foo").create();
GitlabIssue updatedIssueFoo = issueFoo.withDescription("desc").update();
GitlabIssue deletedIssueFoo = updatedIssueFoo.delete();
```

Based on our observation on the client code, the second alternative is more concise and also it avoids making the caller do the work twice, that is, update fields once and execute the update operation again. Also, we thought it is better to narrow down the scope where users can access newly generated component instances to prevent failure in advance. If not doing this, the user may want to create some invalid component instance (e.g., creating an issue without required fields like projectId). Thus, the second alternative is adopted.

4.2 Immutability Issues

We decide to make all component classes mutable. Although immutable objects are simple, inherently thread-safe, we benefit little from making them immutable in current settings. Firstly, component objects are mostly operated by single-threaded users and sessions and are scarcely shared. Secondly, making them immutable induces high overhead when constructing new objects from old ones. Updating multiple fields of component objects is a common operation. We should not create a separate object for each updated field when the identity of the component stays unchanged. Thus, we keep them mutable.

Although we didn't make component classes immutable, we minimize the mutability as much as possible. We intentionally finalized all component classes and made all fields private. We only exposed setters for modifiable attributes and made other fields final. The accessibility of existing component objects is allowed only from their parent component.

4.3 Design for Usability and Quality

Another major design decision is to **avoid long and inconsistent parameter lists and use Builder Pattern instead**.

```
// Say if we already have a client and want to create a project.

// The original API requires 10 parameters in total to create a project. (Ugly)
GitlabProject project = api.createProject("name", namespaceId, true, false,
                            true, false, true, true, visibility, "importUrl");

// The fixed API can build new projects with only necessary parameters.
GitlabProject project = client.newProject("projectName")
                   .withDescription("foo")
                   .create();
```

Note that the API client should adopt Builder Pattern too because it has required an argument endpoint and all other fields are optional. Rather than Static Factory Pattern, it supports optional parameters better and also the naming can be incomprehensive if using static factory methods.

```
// Builder Pattern.
GitlabAPIClient client = new GitlabAPIClient
                            .Builder("gitlab.com")
                            .withAccessToken("token")
                            .withApiNamespace("api/v4")
                            .build();

// Static Factory Pattern. (Worse)
GitlabAPIClient client = GitlabAPIClient.getClientWithAccessToken("gitlab.com",
"token");
```

In addition, **we decided to keep rigorous naming consistency**. Getters for components' individual fields are named with `getXXX()` except for getters for booleans values are named as `isXXX`. Setters for Gitlab component classes and their corresponding query classes are named as `withXXX()` Query method for a single component is named after `getXXX()` and for multiple possible component results is named as `getXXXsQuery()`. Accessors for builders for components are named after `newXXX()`. CRUD operations for each component are `create()`, `query()`, `update()`, and `delete()`. There is no exception to this set of naming rules.

Also, we provided a method `toJson()` for every component class, which saves users from generating JSON body by themselves.

4.4 Support Query

The original API fails to provide what's common to do for Gitlab API users, which is to query with parameters for any Gitlab component. We decided to support this feature. Taking user query as example, we probably need to **support querying with both required and optional parameters**. Consequently, we developed a query class for each Gitlab component individually and expose the builder for query from its parent level. For example, if we want to query commits, users can only get the query builder for commit query from project and specify their query limits or paginations, and thus can perform the query.

```
// Say if we already have a project and we want to query all the commits since
yesterday and after git ref named "master".
List<GitlabCommit> commits = project.getCommitsQuery()
                                    .withSince(LocalDateTime.now().minusDays(1))
                                    .withRefName("master")
                                    .query();
```

The original API has potential performance issues because it allows searching without limiting searching scope and range. We placed conservative limits on the size of query results via **passing Pagination into each query operation to limit searching range**.

```
// Say if we already have a project and we want to query all issues assigned to
some user with userId.

// We can specify pagination explicitly.
Pagination pagination = Pagination.of(2, 20);
List<GitlabIssue> issues = project.getIssuesQuery()
                                  .withAssigneeId(userId)
                                  .withPagination(pagination)
                                  .query();

// Or, a default pagination is applied for query in the background, which is
equivalent to using default pagination.
Pagination defaultPagination = Pagination.getDefaultPagination();
List<GitlabIssue> issues = project.getIssuesQuery()
                                 .withAssigneeId(userId)
                                 .query();
```

4.5 Exception Design

As briefly mentioned in section 3, the original API throws IOException whenever a http request is made. We instead wrapped exceptional conditions with unchecked GitlabException with comprehensive failure-capture information. If the error is explicitly specified by response from Gitlab REST API, we directly use it as output. Otherwise, if a user's request is invalid (e.g., insufficient information to create something), the exception would indicate that it cannot be serialized or parsed. If the request fails to reach the server-side, this would throw a timeout error. If the response information format is broken, this would indicate that responses can not be parsed. Instead of only specifying error code in exceptions, we provided error messages that users can understand readily and use for debugging.

Regarding whether to make the GitlabException checked or unchecked, we think the client code will be redundant and ugly with checked GitlabException because it forces users to use try-catch blocks. Furthermore, even if catching the exception, users can do little except for printing out the exception information or error code.

```
// If using GitlabException as checked exception
try {
    GitlabProject project = client.newProject("example-project").create();
    project.withDescription("desc").update();
    project.delete();
} catch (GitlabException e) {
    // deal with the exception, probably can only print out error information
    e.printStackTrace();
}

// If using GitlabException as unchecked exception
GitlabProject project = client.newProject("example-project").create();
project.withDescription("desc").update();
project.delete();
```

5. Test Coverage

End-to-end and unit tests are written for each of the Gitlab components and execute on Gitlab. The scenarios included in the tests are as follows:

1) Sequential CRUD - create, read, update, read, delete, read
2) Sequential CRD - create, read, delete, read
3) Duplicate create - create, create, read, delete, read
4) Duplicate delete - create, read, delete, delete, read
5) Duplicate update - create, read, update, update, read, delete, read
6) Update on non-existent object - update, update
7) Query - query all instances on Gitlab/under current project
8) Change of state - the state of merge requests and issues change from opened to closed
9) equals

However, for GitlabCommit and GitlabUser, we did not test the functionalities of create, update and delete because considering the reality that only administrators have the permission to create or modify these instances, our implementation does not provide these functionalities for the two components. **All tests are passed**.

6. Future Work

6.1 An Alternative to Pagination

Right now when a user issues a query to get a list of GitLab components (e.g. GitlabProject), by default only the components in the first page will be returned since we only issue one HTTP request to the Gitlab API endpoint per query using Pagination. However, letting the user manually maintain and increase the page number until the resources are exhausted is not ideal. In most scenarios, the user only wants to specify an offset (the number of items to be skipped, usually will be 0) and a limit (how many items to be returned at most, e.g. top 100 items) and would like the API handle the paginations for them. Therefore, in the future we would like to add 2 additional APIs: offset(int n) and limit(int n) to the GitlabQuery class so the users can have a better way to control how to retrieve a list of components without worrying about pagination themselves.

```
// get the most recent 10 projects I owned
List<GitlabProject> projects = client.getProjectsQuery()
                                     .withOwned(true)
                                     .withSort("desc")
                                     .limit(10)
                                     .query();
```

6.2 A Larger Scope of Gitlab APIs

Due to the limited time we have to finish this project, we can only fix some of the most commonly used APIs mentioned above. Considering there are still many APIs provided by GitLab, in the future we should incorporate as many as APIs as possible. For example, Group APIs (similar to Organizations in GitHub) are quite important since it provide ways to manage group owned projects instead of individual user, which may be useful for the team collaboration as well as corporate settings

6.3 Admin Access

Currently we only support API calls from a normal user's perspective. It will be beneficial to support APIs from an administrator(super user)'s perspective as well. It's used quite often in corporate settings that system administrators build and maintain an internal GitLab Service. In this way, support admin access will significantly help these administrators to manage all internal users, groups as well as projects programmatically in Java.

6.4 More Exhaustive Testing

Our test cases right now directly send requests to the GitLab public API endpoint and assert the response according to the expected behavior. However, this highly relies on the network call which is not suitable for fine-grained testing and also can not test cases for admin access. Therefore, we can first try to test using Mockito or other third party libraries that can mock the HTTP responses. Second, we can set up our testbed using GitLab docker image. This will make our tests independent from the actual public GitLab endpoint and also enables us to test admin access since we will be the admins in the GitLab we created in the docker container. Both methods will enable our tests to run without the actual network support and make our tests reproducible and consistent across all platforms.

6.5 Representation of Component Parameters in Gitlab API

There are a couple of parameters in the Gitlab Web API that require specific constant string values. For example, sort will take string value desc and asc for sorting the descending and ascending. We could introduce specific Enum types to keep them fixed and avoid users from typos and misuses of such parameters.

7. Contributions

All of our team members actively participated in designing the high level architecture of our new API and writing this final report. After setting up the interfaces, Xinyi and Ben were in charge of the implementation and documentation of the API while Rong and Xuanxuan wrote the tests and the client code. Although we separated the team into two parts working on different aspects of this project, we met regularly to discuss and refine our design.

8. Javadoc

https://apiteam4.gitlab.io/f20-project-team4/