

Git as Subversion

User manual

Artem Navrotskiy

Git as Subversion: User manual

Artem Navrotsky

Publication date 2016

Copyright © 2014-2016 Artem Navrotsky

Table of Contents

1. About project	1
What is it?	1
What is project goal?	1
How does it work?	1
Where is the Subversion data stored?	1
How does commit works?	2
Unlike other solutions	2
GitHub Subversion support	2
SubGit	3
Subversion repository and git svn	3
Features	3
What is already there?	3
What is lacking?	4
Technical limitations	4
2. Prerequisites	5
Recommendations for Subversion-client	5
Recommendations for Git-client	5
LFS for Git users using SSH protocol	5
LFS for Git users using HTTP protocol	5
Recommendations for Git-repositories	5
File .gitattributes	5
3. Installation	7
Quick start	7
Installation on Debian/Ubuntu	7
Package git-as-svn	7
Used directories	8
Package git-as-svn-lfs	8
Build from source code	9
4. GitLab integration	10
Recommended GitLab patches	10
GitLab intergration points	10
Adding a SVN-link to GitLab interface	10
Configuration file example	11
5. SVN Properties	13
File .gitignores	13
File .gitattributes	14
File .tgitconfig	14
6. API	15
Interface description	15

Chapter 1. About project

What is it?

Git as Subversion (<https://github.com/bozaro/git-as-svn>) — Subversion-server implementation (svn protocol) for Git-repositories.

This project allows to work with Git-repository using Subversion console client, TortoiseSVN, SvnKit and other similar tools.

What is project goal?

The project is designed to allow you to work with the same repository as Git, Subversion and style.

Git style

The basic idea is that the developer works in the local branch. His changes do not affect the work of other developers, but nonetheless they can be tested on CI farm, review by another developer and etc.

This allows each developer to work independently, as best he can. He can change and saving intermediate versions of documents, taking full advantage of the version control system (including access to the change history) even without network connection to the server.

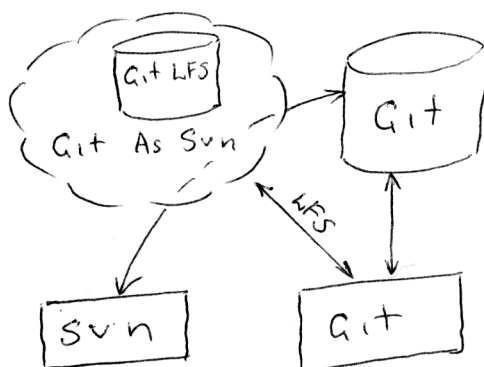
Unfortunately, this approach does not work with not mergeable documents (for example, binary files).

Subversion style

The use of a centralized version control system is more convenient in the case of documents do not support the merge (for example, with binary files) due to the presence of the locking mechanism and a simpler and shorter publication cycle changes.

The need to combine Git and Subversion style work with one repository arises from the fact that different employees in the same project are working from fundamentally different data. If you overdo, you Git programmers, and artists like Subversion.

How does it work?



Where is the Subversion data stored?

To represent Subversion repository need to store information about how Subversion-revision number corresponds to which Git-commit. We can't compute this information every time on startup, because first **git push --force** change revision order. This information stored persistent in git reference `refs/git-as-svn/*`. In particular because it does not require a separate backup Subversion data. Because of this separate backup Subversion data is not necessary.

Also part of the data necessary for the Subversion repository, is very expensive to get based on Git repository.

For example:

- the revision number with the previous change file;
- information about where the file was copied;
- MD5 file hash.

In order not to find out their every startup, the data is cached in files. The loss of the cache is not critical for the operation and backup does not make sense.

File locking information currently stored in the cache file.

How does commit works?

One of the most important parts of the system — to save the changes.

In general, the following algorithm:

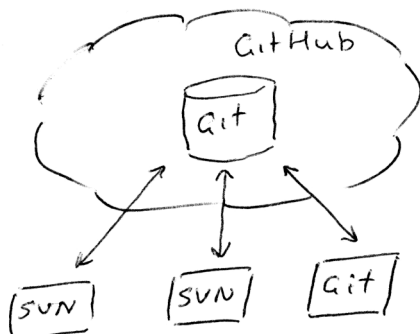
1. At the moment the command **svn commit** client sends to the server of your changes. The server remembers them. At this point comes the first check the relevance of customer data.
2. The server takes the branch HEAD and begins to create new commit on the basis of client received delta. At this moment there is yet another check of the relevance of customer data.
3. Validating svn properties for changed data.
4. The server tries to push the new commit in the current branch of the same repository via console Git client. Next, the result of a push:
 - if commits pushed successfully — loading the latest changes from git commits and rejoice;
 - if push is not fast forward — load the latest changes from git commits and go to step 2;
 - if push declined by hooks — inform the client;
 - on another error — inform the client;

Thus, through the use console Git client for push, we avoid the race condition pouring directly change Git repository, and get the native hooks as a nice bonus.

Unlike other solutions

The problem of combining Git and Subversion work style with a version control system can be solved in different ways.

GitHub Subversion support

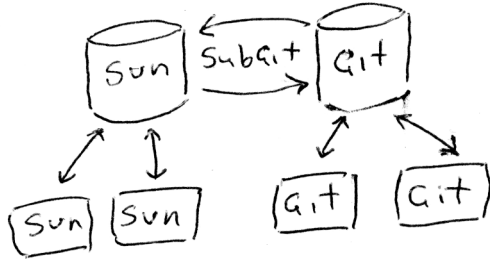


This is probably the closest analogue.

The main problem of this implementation is inseparable from GitHub. Also, all of a sudden, this implementation does not support Git LFS.

In the case of GitHub it is also not clear where the stored mapping between Subversion-revision and Git-commit. This can be a problem when restoring repositories after emergency situations.

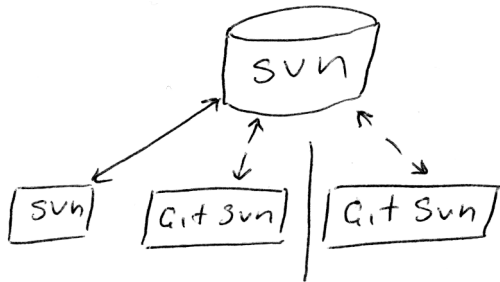
SubGit



Web site: <http://www.subgit.com/>

Quite an interesting implementation which supports master-master replication with Git and Subversion repositories. Thereby providing synchronization of repositories is not clear.

Subversion repository and git svn



This method allows you to use Git with Subversion repository, but using a shared Git repository between multiple developers very difficult.

At the same time, the developer has to use a specific command-line tool for working with the repository.

Features

This implementation allows the majority of Subversion-users to work without thinking about what they actually use Git-repository.

What is already there?

- You can use at least the following clients:
 - Subversion console client;
 - TortoiseSVN;
 - SvnKit.
- Supported subversion operations:

- svn checkout, update, switch, diff
- svn commit
- svn copy, move¹
- svn cat, ls
- svn lock, unlock
- svn replay (svnsync)
- Git LFS support;
- Git submodules supported;²
- LDAP authorization;
- GitLab integration.

What is lacking?

- Large gaps in the documents;
- You can only access one branch from Subversion.

Technical limitations

- It is impossible to change svn properties by Subversion client;
- Empty directories is not allowed.

Chapter 2. Prerequisites

Recommendations for Subversion-client

For automatic Subversion properties set to added files and directories used the inherited properties feature.

This feature is supported since Subversion 1.8.

If you are using TortoiseSVN and `bugtraq: *` properties, then you need to use TortoiseSVN 1.9 or later.

Recommendations for Git-client

LFS for Git users using SSH protocol

Git-client to obtain LFS authentication data by executing on server via SSH `git-lfs-authenticate` command.

This request can be run very often. The establish SSH connection spent a lot of time (about 1 second).

To reduce SSH connection establish time, you can enable re-use of SSH connections.

You can enable SSH session reuse on Linux by command:

```
#!/bin/sh
echo "Host *"
    ControlMaster auto
    ControlPath ~/.ssh/controlmasters/%r@%h:%p
    ControlPersist 10m
" > ~/.ssh/config
mkdir ~/.ssh/controlmasters
chmod 700 ~/.ssh/controlmasters
```

LFS for Git users using HTTP protocol

Git client can ask login and password for LFS storage for each file.

To avoid this, it is necessary to enable Git password caching.

You can enable password cache by command:

```
git config --global credential.helper cache
```

By default passwords are cached for 15 minutes.

You can change cache lifetime by command:

```
git config --global credential.helper 'cache --timeout=3600'
```

Более подробная информация доступна по адресу: <https://help.github.com/articles/caching-your-github-password-in-git/>

Recommendations for Git-repositories

File .gitattributes

By default Git using native line ending for text files.

To keep text files original content by default you need add to begin of file `.gitattributes` line:

```
* -text
```

Chapter 3. Installation

Quick start

To try Git as Subversion you need:

1. Install Java 8 or later;
2. Download archive from site <https://github.com/bozaro/git-as-svn/releases/latest>;
3. After unpacking the archive change working path to the uncompressed directory and run the command:

```
bin/git-as-svn --config doc/config-local.example --show-config
```

This will start Git as Subversion server with following configuration:

1. The server is accessible via svn-protocol on port 3690.

You can check server with command like:

```
svn ls svn://localhost/example
```

2. To access the server, you can use the user:

Login: test

Password: test

3. Cache and repository will be created in build directory:

- `example.git` — repository directory, accessible via svn-protocol;
- `git-as-svn.mapdb*` — cache files for expensive computed data.

Installation on Debian/Ubuntu

You can install Git as Subversion repository on Debian/Ubuntu using the commands:

```
#!/bin/bash
# Add package source
echo "deb https://dist.bozaro.ru/ debian/" | sudo tee /etc/apt/sources.list.d/dist.boza
curl -s https://dist.bozaro.ru/signature.gpg | sudo apt-key add -
# Install package
sudo apt-get update
sudo apt-get install git-as-svn
sudo apt-get install git-as-svn-lfs
```

Package git-as-svn

This package contains the Git as Subversion.

After you install Git as Subversion is run in daemon mode and is available on the svn-protocol on port 3690. The daemon runs as `git` user.

To access the server, you can use the user:

Login: test

Password: test

You check configuration with command like:

```
svn ls --username test --password test svn://localhost/example/
```

Used directories

This package by default is configured to use the following directories:

/etc/git-as-svn

This directory contains configuration files.

/usr/share/doc/git-as-svn

This directory contains this documentation to the installed version.

/var/git/lfs

This directory contains configuration files.

It must be writable for the user `git`.

/var/git/repositories

This directory is used by default to store the Git-repositories.

Repositories must be writable for the user `git`.

/var/log/git-as-svn

This directory is used to record log files.

It must be writable for the user `git`.

Log rotation configuration can be changed by `/etc/git-as-svn/log4j2.xml` file.

/var/cache/git-as-svn

This directory is used to store the Git as Subversion cache.

It must be writable for the user `git`.

The loss of the contents of this directory is not critical for operation and does not entail the loss of user data.

Package git-as-svn-lfs

This package contains the `git-lfs-authenticate` script.

Script `git-lfs-authenticate` is used for provide authentication data for HTTP access to Git LFS server for Git-users working with Git repository by SSH (<https://github.com/github/git-lfs/blob/master/docs/api/README.md>).

This script communicates through a Unix Domain Socket with Git as Subversion.

It send to Git as Subversion user name (`mode = username`) or user identifier (`mode = external`) taken from environment variable. Environment variable name is defined in configuration file via `variable` parameter (default value: `GL_ID`).

To check the settings of the script can be run locally on the server the following command:

```
#!/bin/bash
# Set environment variable defined in configuration file
export GL_ID=key-1
# Check access to repository
sudo su git -c "git-lfs-authenticate example download"
```

Or on the client the following command:

```
#!/bin/bash
ssh git@remote -C "git-lfs-authenticate example download"
```

The output should look something like this:

```
{
  "href": "https://api.github.com/lfs/bozaro/git-as-svn",
  "header": {
    "Authorization": "Bearer SOME-SECRET-TOKEN"
  },
  "expires_at": "2016-02-19T18:56:59Z"
}
```

Build from source code

The project was originally designed for assembly in Ubuntu.

To build from the source code you need to install locally:

1. Java 8 (openjdk-8-jdk package);
2. xml2po (gnome-doc-utils package) — required for build reference files;
3. protoc (protobuf-compiler package) — required for build API.

You can build distribution by command:

```
./gradlew assembleDist
```

Distribution files build in folder: build/distributions

Chapter 4. GitLab integration

Recommended GitLab patches

For integration with GitLab install the following patches on GitLab:

- [#230 \(gitlab-shell\)](#): Add git-lfs-authenticate to server white list (merged to 7.14.1);
- [#237 \(gitlab-shell\)](#): Execute git-lfs-authenticate command with original arguments (merged to 8.2.0);
- [#9591 \(gitlabhq\)](#): Add API for lookup user information by SSH key ID (merged to 8.0.0);
- [#9728 \(gitlabhq\)](#): Show "Empty Repository Page" for repository without branches (merged to 8.2.0).

GitLab intergration points

There are some intgeration points with GitLab:

- The list of repositories

Git as a Subversion automatically retrieves the list of repositories on startup via the GitLab API.

Further, this list is updated by System Hook, which is registered automatically.

- Authorization and authentication of users

For authenticating users and repository permission control is also used GitLab API.

- Git Hooks

When you commit using Git as Subversion the GitLab hooks are executed. These hooks, in particular, allow to see information about new commits without delay via the GitLab WEB interface.

GitLab Hooks require GitLab user ID information (GL_ID environment variable) received at user authorization.



Important

Because of this, in the case of integration with GitLab user authentication must be via GitLab.

- Git LFS

In the case of GitLFS need to specify the path to GitLab LFS storage.

GitLab from version 8.2 uses single LFS-files storage shared between all repositories. Files are stored in a separate directory as raw data.

Integration with LFS repository GitLab occurs at the file level. GitLab API is not used.

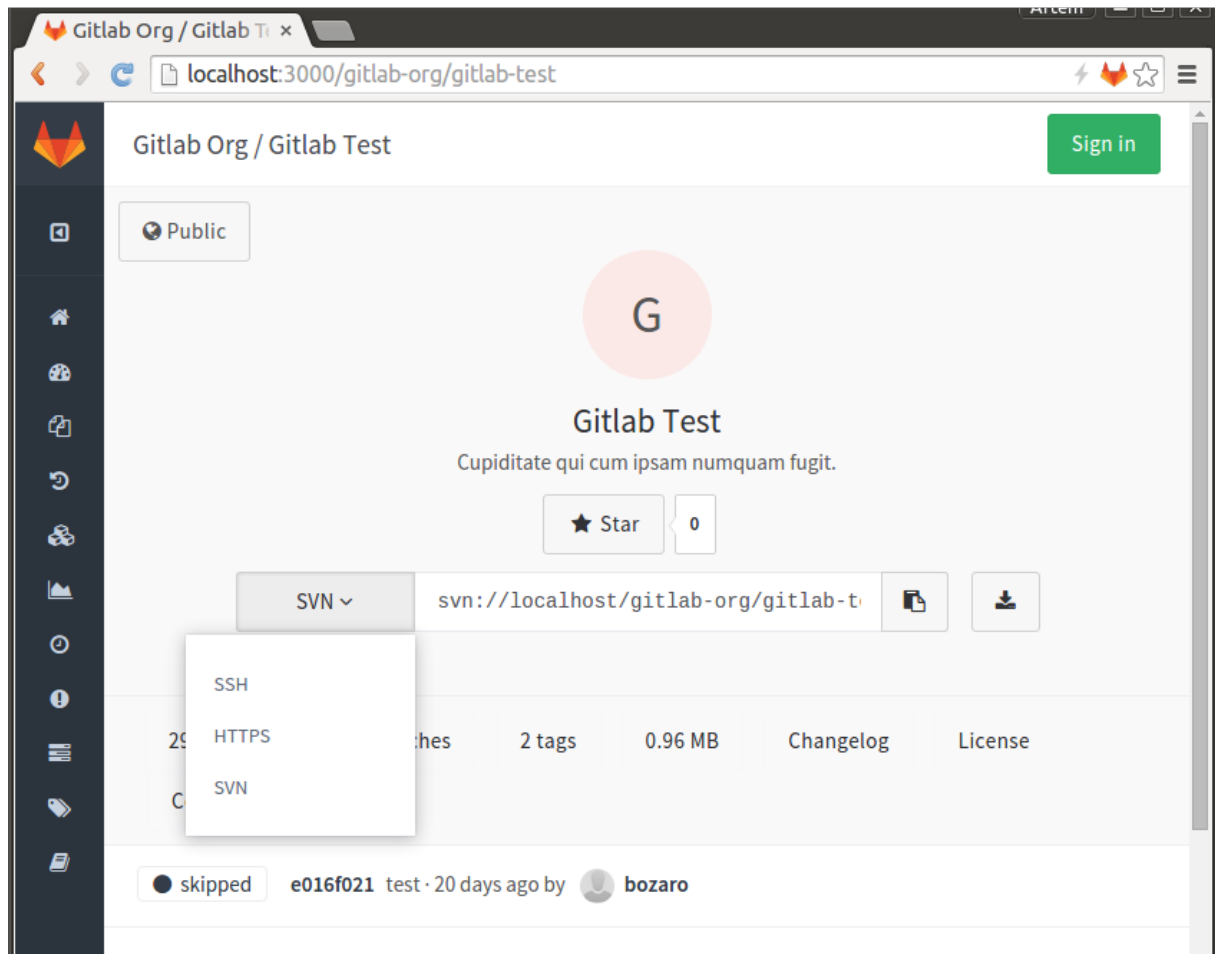
- Git LFS authorization for SSH users

Unfortunately, GitLab doesn't provide the git-lfs-authenticate script, which is responsible for SSO authorization SSH-user Git on the server LFS. To configure this script, see the section called "Package git-as-svn-lfs".

Adding a SVN-link to GitLab interface

To add a SVN-link to GitLab interface need to take latest commit of branch https://github.com/bozaro/gitlabhq/commits/svn_url.

Example of SVN-link in GitLab interface



Configuration file example

```

1 !config:
2 realm: Example realm
3 compressionEnabled: true
4
5 # Use GitLab repositories
6 repositoryMapping: !gitlabMapping
7   path: /var/opt/gitlab/git-data/repositories/
8   template:
9     branch: master
10    renameDetection: true
11
12 # Use GitLab user database
13 userDB: !gitlabUsers {}
14
15 shared:
16   # Web server settings
17   # Used for:
18   # * detecting add/remove repositories via GitLab System Hook
19   # * git-lfs-authenticate script (optional)
20   - !web
21     baseUrl: http://git-as-svn.local/
22     listen:

```

```
23     - !http
24       host: localhost
25       port: 8123
26       # Use X-Forwarded-* headers
27       forwarded: true
28     # GitLab LFS server
29     - !lfs
30       # Secret token for git-lfs-authenticate script
31       # token: secret
32       path: /mnt/storage/lfs-objects
33       saveMeta: false
34       compress: false
35       layout: GitLab
36     # GitLab server
37     - !gitlab
38       url: http://localhost:3000/
39       hookUrl: http://localhost:8123/
40       token: qytzQc6uYiQfsoqJxGuG
41
```

Chapter 5. SVN Properties

The main `svn` properties trouble that they should be maintained in the synchronous state between Git and Subversion.

Because of this arbitrary `svn` properties is not supported. To value `svn` propertiescode> correspond Git-view, they are generated on the fly based on the repository content.

Wherein:

- the commit verifies that `svn` properties file or directory exactly match what should be according to the data repository;
- Subversion does not tool allows you to change most of the properties (exception: `svn:executable`, `svn:special`);
- if a file affects the `svn` properties other files after changing it `svn` properties of the files in the same change.



Important

For user convenience Git as Subversion is actively using the inherited properties.

This feature require to use the client Subversion 1.8 or later.

Otherwise there will be problems with the `svn` properties for new files and directories.

File .gitignores

This file affects the property `svn:ignore` and `svn:global-ignores` for the directory and its subdirectories.

For example, a file in the directory `/foo` with the contents:

```
.idea/libraries
*.class
*/build
```

Mapped to properties:

- for directory `/foo`:
`svn:global-ignores: *.class`
- for directory `/foo/*`:
`svn:ignore: build`
- for directory `/foo/.idea`:
`svn:ignore: libraries build`



Important

For Subversion has no way to make an exception for directories, as a result, for example, the rules of `/foo` (file or directory foocode>) and `/foo/` (directory foo) in Subversion will work the same way, though to Git they have different behavior.

Terms like "all but" not supported on mapping to the `svn:global-ignores` property.

File .gitattributes

This file affects the properties of the `svn:eol-style` and `svn:mime-type` files from this directory and `svn:auto-props` from the directory itself.

For example, a file with contents:

```
*.txt          text eol=native
*.xml          eol=lf
*.bin          binary
```

Add property to the directory `svn:auto-props` with the contents:

```
*.txt = svn:eol-style=native
*.xml = svn:eol-style=LF
*.bin = svn:mime-type=application/octet-stream
```

And files in this directory:

- for suffix `.txt` add property `svn:eol-style = native`
- for suffix `.xml` add property `svn:eol-style = LF`
- for suffix `.bin` add property `svn:mime-type = application/octet-stream`

File .tgitconfig

This file only changes the properties of the directory in which it is located.

Properties are mapped one-to-one, for example, a file with the contents:

```
[bugtraq]
url = https://github.com/bozaro/git-as-svn/issues/%BUGID%
logregex = #(\d+)
warnifnoissue = false
```

It will be converted to properties:

- `bugtraq:url = https://github.com/bozaro/git-as-svn/issues/%BUGID%`
- `bugtraq:logregex = #(\d+)`
- `bugtraq:warnifnoissue = false`

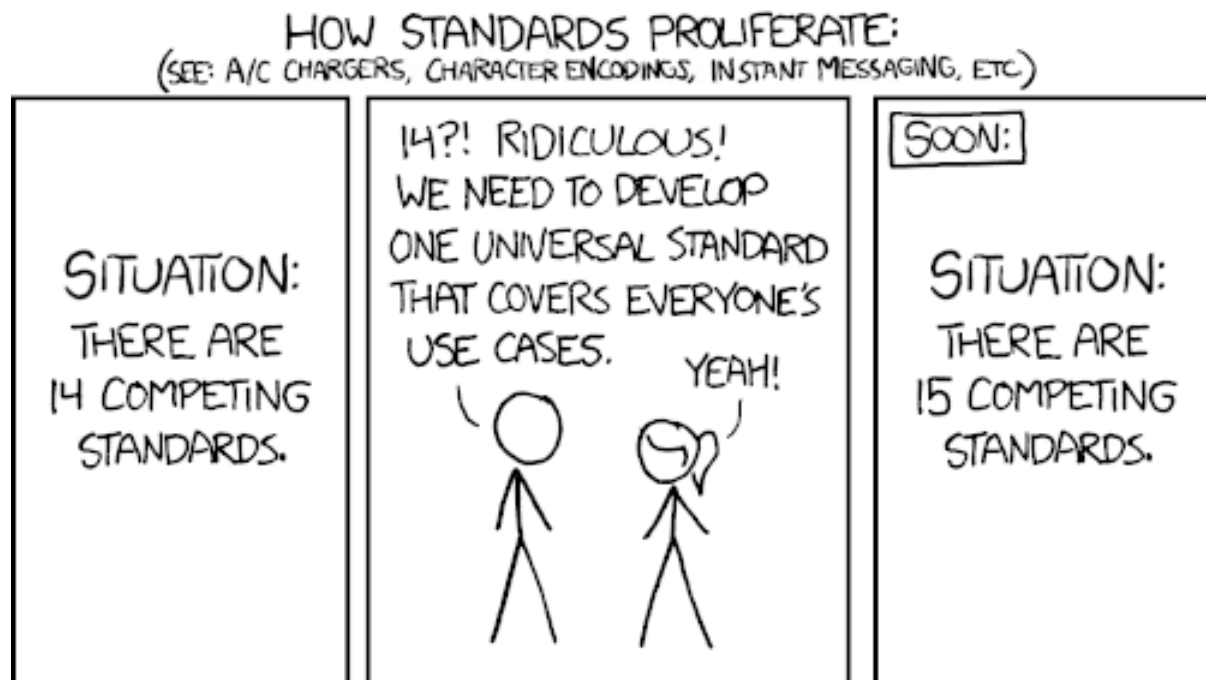


Important

If you use bugtraq svn properties, it is highly recommended that you use TortoiseSVN 1.9 or later.

Otherwise TortoiseSVN will attempt to set these parameters for all newly created directories instead of use inherited properties.

Chapter 6. API



<https://xkcd.com/927/>

Interface description

API is implemented according to the following requirements:

- Schema-first API definition.

This allows the use schema as the documentation, and in some cases, to generate the code for the work with the API.

- Client implementation must be trivial on any programming language.

As a result, for API scheme has been chosen Protocol Buffers (<https://developers.google.com/protocol-buffers/>).

Since Protocol Buffers does not have a native RPC implementation, used custom RPC implementation based on HTTP protocol.

To call the method, a request is sent to a URL of the form:

```
http://somehost/<repository>/<service>/<method>.<format>
```

Wherein:

<repository>

The name of the repository, for which the method is called API.

Don't used for server-wide API.

<service>

Lower case service name from .proto file.

<method>

Lower case method name from .proto file.

<format>

Message serialization format.

Supported formats:

- bin (application/x-protobuf)

Binary Protocol Buffers serialization.

- json (application/json)

JSON serialization. Allows to implement API without Protocol Buffers library.

- xml (application/xml)

XML serialization. Allows to implement API without Protocol Buffers library.

- txt (text/plain)

Text serialization. It useful for invoke some methods from Internet browser.

The RPC-call argument may be transmitted via POST-request body.

Scalar top-level attributes also can be passed via URL parameters.