Publish and Share

Extending Publish and Share

Version 1.0
2019-08-07

Email: contact@its4land.com
Web: https://its4land.com/

Table of Contents

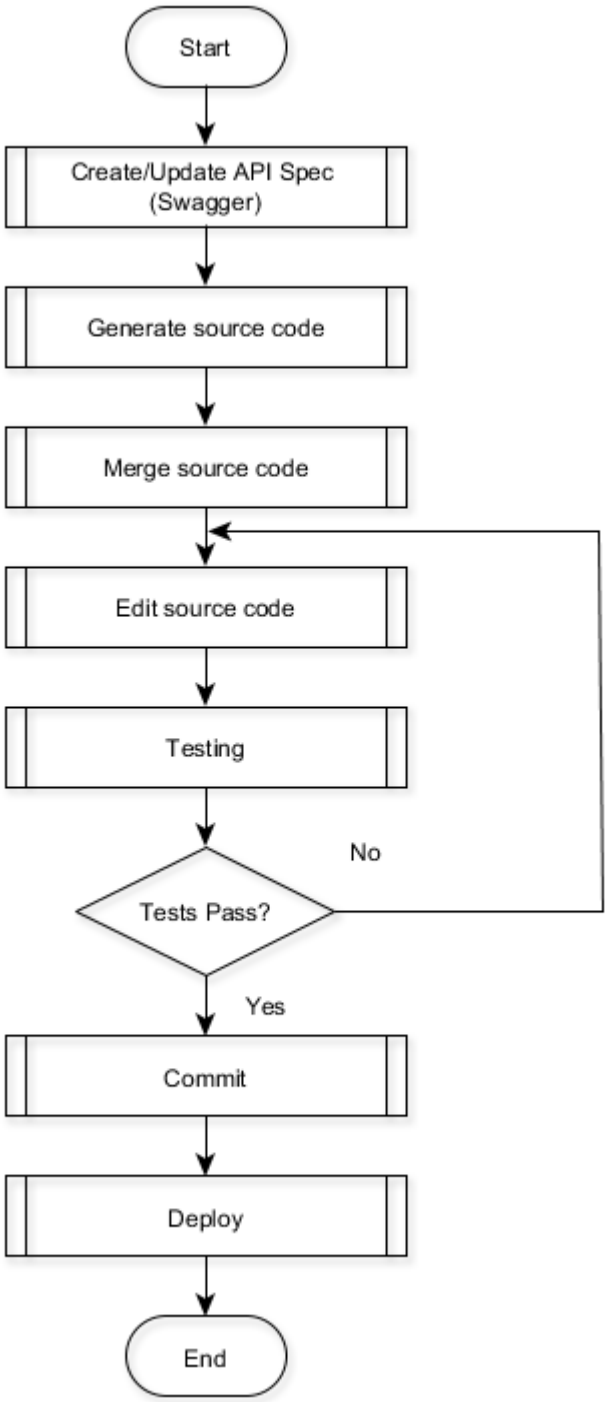
Introduction

This document describes a general development workflow for the Publish and Share (PuS) platform Public API. The intended audience for this manual are software developers aiming to extend customize the Publish and Share platform. These include functionalities such as adding their own API endpoints or using alternate cloud providers. Such users will require knowledge of the OpenAPI specification and programming experience in JavaScript on the NodeJS runtime in order to work with the source code.

Development generally proceeds in the sequential manner as seen in the flowchart below. The following sections describe technical details involved in each step.

Note: All steps assume that a copy of the Its4land repository has been cloned using git

```
git clone git+ssh://<path-to-repository>
```


Start
Create/Update API Spec
(Swagger)
Generate source code
Merge source code
Edit source code
Testing
Tests Pass?
No
Yes
Commit
Deploy
End

1. Create/Update API specification (Swagger)

The API for the Publish and Share is specified in Swagger (aka OpenAPI v2.0 - https://swagger.io). The API can be specified either in YAML or JSON as markup languages, with the formats being interchangeable. An online editor is available to load and edit Swagger specifications, preview documentation and checks for syntax on the fly (http://editor.swagger.io/#). The editor can also be used to load and convert between markup formats. Additionally it is possible to generate code for the API server or client in supported languages.

The specification for the PuS public API can be found in the following path:

```
<path-to-repository>/PublicAPI/swagger_publicAPI.json
```

2. Generate Source Code

The next step after the API has been created/modified is to generate the source code for the target platform (NodeJS on the server, Javascript on the client in the case of PuS).

While the code generation can be done directly in the Swagger editor, it uses an older version of the generator resulting in slight differences in generated code. It is *recommended to use the patched version of the code generator* available in the repository as a Java JAR file, which can be found in the following location:

```
<path-to-
repository>/ExperMaps/server/node/plugins/publishAndShare/unittests/tools/swagg
er-codegen-cli-modified.jar
```

Code Block 1 Location of patched Swagger code generator

The patched version differs from the online hosted version in the following ways:

- Does not override comma-separated parameters values by multiple parameter instances (https://swagger.io/specification/v2/#parameterCollectionFormat) (Server)
- Adds parameter to return the entire HTTP response instead of just the response body when using Javascript with Promises (Client)
- Marks testcases as pending instead of empty/fail + whitespace formatting changes, in generated testcases (Client)

Note: Running the JAR file requires Java (v8 or higher) to be installed and available in PATH environment variable. Java can be downloaded here https://www.java.com/

Generate Server Code

Generation of the server code *only generates a source code template.* The functionality still needs to be implemented manually.

Server code for the NodeJS platform can be generated by running the following command

```
# Replace the <parameters> with corresponding paths
java -jar <path-to-jar-file> generate -l nodejs-server -i <path-to-swagger-
file> -o <output-directory>
```

Generate Client Library Code

To generate a Javascript client library using Promises (**recommended**) for asynchronous API calls , first create a JSON configuration file with the following content

```
{
    "usePromises": true
}
```

Code Block 2 config.json

Next, run the following command

```
# Replace the <parameters> with corresponding paths
java -jar <path-to-jar-file> generate -l javascript -c config.json -i <path-to-swagger-file> -o <output-directory>
```

Code Block 3 Client library with Promises

One can also generate a client library using the old callbacks style (**not recommended**) in the following manner

```
# Replace the <parameters> with corresponding paths
java -jar <path-to-jar-file> generate -l javascript -i <path-to-swagger-file> -o <output-directory>
```

Code Block 4 Client library using callbacks

3. Merging Source Code

The development cycle is iterative i.e `Change API Spec → Generate Code → Merge with existing code` .... (repeat)

Merging is a crucial step to add new changes in the API specification to the code whilst retaining implemented changes.

Let us consider two directories containing existing code and the newly generated code

1. OLDDIR - This directory contains existing generated code, which has been modified to implement API functionality. This is usually:
    - `<path-to-repository>/ExperMaps/server/node/plugins/publishAndShare/swagger-api` - For the server code
    - `<path-to-repository>/ExperMaps/server/node/plugins/publishAndShare/unittests` - For the client code
2. NEWDIR - Contains newly generated source code following the steps described in the previous section

Merging makes use of an external diff-ing tool such as WinMerge (http://winmerge.org/) to compare corresponding individual files and merge newly added changes, while retaining existing modifications.

An example of a typical merging process is as follows (for the server side code):

- Compare `OLDDIR/service/ProjectsService.js` against `NEWDIR/service/ProjectsService.js`
- Add/subtract any newly made changes to `OLDDIR/service/ProjectsService.js` . These include modifications such as :
    - addition or removal of an endpoint or operation
    - addition or removal of a parameter
    - change of parameter type or name
    - change in documentation / description
- Functions which have already been implemented in OLDDIR/service/ProjectsService.js will be retained, but edited manually to reflected changes in their parameters

In addition to repeating the above process for each of the changed files, we need to:

- Add new files from NEWDIR to corresponding location in OLDDIR
- Remove files, if any, that were deleted in NEWDIR as part of API changes

- Compare the modified API specification itself (this file is generated and slightly different from the original specification):
    - `<path-to-repository>/ExperMaps/server/node/plugins/publishAndShare/swagger-ui/swagger.yaml`, with
    - `NEWDIR/api/swagger.yaml`
    - merge the changes.

Merging for the generated client code is similar. The client library is made use of in unittests: `<path-to-repository>/ExperMaps/server/node/plugins/publishAndShare/unittests`

To merge the client code:

- Replace the `src` and `docs` directories in the 'unittests' directory with the newly generated code. There is no need to compare here
- Merge the files in the existing `test` directory, with the newly generated one such that test cases that were implemented are retained, but edited to reflect new changes to the API spec.

4. Edit Source Code (aka Implementation)

On the server side:

- the generated code is simply a template
- the functionality needs to be implemented manually
- the idea is to make the minimum amount of changes to the generated code. Therefore the core functionality is implemented in separate modules under:
  `<path-to-repository>/ExperMaps/server/node/plugins/publishAndShare/implementation`

On the client side, the generated code for unittest is a template which needs to be implemented. The client libraries themselves can be used as is.

5. Testing

Once the source code code has been edited, it is important to test the changes to check for correct functionality. Testing can be done manually or via the unittests suite.

Manual Testing

This makes use of an application that is able to make HTTP requests such as a browser, convenient GUI applications such as Postman (https://www.getpostman.com) or CLI utilities such as Curl (https://curl.haxx.se). To manually test the functionality:

- Start the server

```
node <path-to-repository>/ExperMaps/server/node/bin/em.js
```

Code Block 5 Start server

- Make an HTTP request to an endpoint by passing the appropriate parameters
- Validate the response body and status code

Unittests

Unittests follow the test driven development (TDD) or behavior driven development (BDD) paradigms and run automated tests to verify the functionality of each individual endpoint. An advantage of utilizing unittests is verifying that code edits do not affect other parts of the platform which were running successfully and preventing regressions.

We make use of the Mocha test framework (https://mochajs.org), for the simple reason that it is the framework used by the code generator. Each endpoint API path is captured in a test *spec* file e.g. `ProjectsApi.spec.js` contains tests pertaining to the Projects endpoint.

Unittests can be run in entirety or selectively. A consolidated report is shown at the end of a run.

```
cd <path-to-repository>/ExperMaps/server/node/plugins/publishAndShare/unittests

# Run the entire testsuite
npm test

# Run tests under projects api only
npm test -- -g 'ProjectsApi'

# Run only a single test with the given Swagger operationId pattern
npm test -- -g 'getProjectsByUID'
```

Code Block 6 Running Unittests

Adding Unittests

- Edit the required `*.spec.js` files under `unittests/test` directory. Use implemented tests for reference
- Individual tests are added as part of the `it(...)` function
- Changing `it(...)` to `xit(...)` marks the test as pending and does not run it
- Assertions are used to test whether an obtained value matches an expected value (https://github.com/Automattic/expect.js). Tests fail if there is no match or if an exception occurs.

Note: As of 2018-12-10, unittests coverage is only partial. Test for a number of endpoints and their operations are yet to be implemented.

6. Committing Code

- While developers are free to commit to their local repository, code should be pushed to the remote origin only when it has been tested and is clean.
- When doing a pull, it is a good idea to do a `git pull --rebase` to append your change to the HEAD of the working branch. This avoids unnecessary merge commits.
- Use short but meaningful commit messages. Ideally, your commit message should be a continuation of the sentence '*Applying this commit will <commit message goes here>*'
- An example commit message would be '*fix GET Projects endpoint response status code*'
- While any number of commits may be used locally, when pushing it is a good idea to group related commits into one, rather than having commits which change only a single line. Using `git rebase --interactive` and *squashing* commits is tremendously useful here. Look it up!
- For any major or breaking change, it is best to create a separate branch and add new features to it. This branch may then be merged with the main development branch when stable enough.