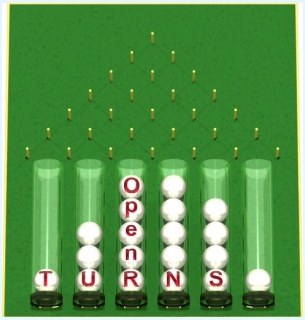
OpenTURNS modules development

Trainer : Sofiane Haddad
Airbus
sofiane.s.haddad@airbus.com

Developers training



OpenTURNS modules development

OpenTURNS modules

Module development - C++

Module development - Pure python module

OpenTURNS modules

OpenTURNS modules: objectives

OpenTURNS is a growing system developed by a small team. A constant problem is to assess the stability of the whole product, and in order to achieve this goal we introduced a notion of module, in order to insulate a core library dedicated to the definition of the abstract data model and to propose all the specific algorithms as optional modules. The core would evolve quite slowly, insuring its robustness, whereas the modules would have a more dynamic development model.
Another key objective is to provide a way to extend the existing platform with functionalities developed by teams that are reluctant to adopt the OpenTURNS development process. Within the module, the development team can adopt any coding rule or programming language he want, as long as the OpenTURNS interface is respected as well as the objects lifecycle.

Current modules - a non exhaustive list

Modules connected to the library and maintained by the consortium :

- otagrum : create a distribution from a Bayesian Network using aGrUM
- otfftw : Fast Fourier Transform algorithm (e.g. for stochastic processes) using FFTW
- otfmi : FMI models manipulation using PyFMI
- otmixmod : build mixtures of a multivariate Normal distribution from a sample
- otmorris : Morris screening method module
- otpmml : manages PMML files for meta-modeling exchanges
- otpod: A module to build Probability of Detection for Non Destructive Testing
- otrobopt: robust optimization
- otsubsetinverse: inverse subset simulation
- otsvm : Support Vector regression and classification with libsvm
- otwrapy : Python wrapper tools
- otsklearn : use OT surrogate models with the scikit-learn estimator API

Oldest modules (because integrated in OT):

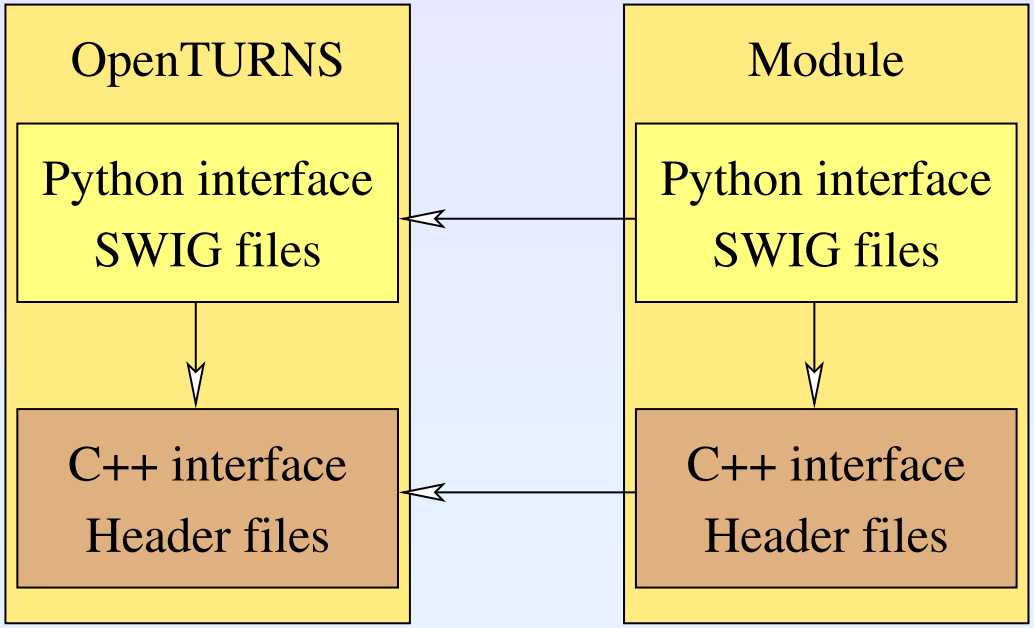
- otlm Linear model with stepwise strategies
- otlhs Optimal LHS (Monte-Carlo & Annealing)
- ...

Current modules - a non exhaustive list

Other modules connected to the library :

- otbenchmark : benchmark problems for reliability and sensitivity analysis
- othdrplot: high density region algorithm for functional outlier detection
- otsurrogate: surrogate models
- otusecases: use cases suitable for OpenTURNS (functions and datasets)
- otmarkov: simulates Markov chains (experimental)
- otsensitivity : sensitivity analysis with density based measures
- otshapley : compute Shapley effects
- otsmt : Simple module that implements bindings from Surrogate Modelling Toolbox (SMT) models to OpenTURNS
- otak : Simple module that implements Active learning Kriging (AK) methods for reliability analysis.
- otkerneldesign : generates designs of experiments based on kernel methods such as Kernel Herding and Support Points
- batman : Statistical analysis for expensive computer codes made easy.
- others : need to reactivate some oldest modules (`otgmm`, `pygosa`, `otfitting`, `otgu`) and others

OpenTURNS modules - C++



The principles
An OpenTURNS module is typically made of two parts:

- a C++ part that uses the OpenTURNS C++ interface in order to provide new specialization or to produce instances of the data model using new algorithms with no OpenTURNS counterpart.
- a Python part, the Python interface of the C++ part, often obtained using SWIG. In this case, this SWIG interface must use the OpenTURNS interface the same way its C++ interface uses the OpenTURNS C++ interface.

Module development

Module development

Step 1: copy and adapt an existing template

- Copy and rename the source tree of an example module (for example the Strange module) from the OpenTURNS source tree. The examples modules are located under the module subdirectory of OpenTURNS source tree:
  `git clone https://github.com/openturns/ottemplate.git MyModule template`
- Set the url of the remote (meaning you already created a repository):
  `git remote set-url origin https://github.com/USERNAME/REPOSITORY.git`
- Adapt the template to your module:
  `./customize.sh MyModule MyModuleClass`
  This command change the module name into all the scripts, and adapt the example class to this new name.

Module development

Step 2: develop the module

- Implement your module. You are free to use the rules you want, but if the final objective of the module is to be integrated in the official release of OpenTURNS, it is wise to adopt the OpenTURNS development process and rules.
- Build your module as usual:

```
mkdir build
cd build
cmake ..  -DCMAKE_INSTALL_PREFIX=INSTALLDIR
-DOpenTURNS_DIR=OPENTURNS_INSTALLDIR/lib/cmake/openturns
make
```

- Create a source package of your module:

  `make package_source`

  It will create a tarball named mymodule-X.Y.Z.tar.gz (and mymodule-X.Y.Z.tar.bz2), where X.Y.Z is the version number of the module.

Module development

Step 3: documentation
Module documentation is very close to the API one:

- Developer guide : Architecture, validation
- SWIG documentation is to be completed (docstrings);
- Examples and API documentation;
- Theory (if needed)

Module development

Step 4: install and test the module

- Check that you have a working OpenTURNS installation, for example by trying to load the OpenTURNS module within an interactive python session:
  ```
  $python
  >>> import openturns as ot
  ```
  and python should not complain about a non existing openturns module.
- Since first step is ok, test your module within python:
  ```
  $python
  >>> import openturns as ot
  >>> import mymodule
  ```
  and python should not complain about a non existing mymodule module.
- You can now use your module as any python module.

Module development

Step 5: Maintenance
Each developer is responsible of his package should maintain it!

- We can rely on continuous integration tool accounting API changes
- He is in charge of bugs (except if underlying bug is in the API)

Exception : modules maintained by the consortium!

Step 6: Packaging
No specific rule concerning the packaging!

- Tarball generation
- Git
- conda...

Exception : modules maintained by the consortium!

Module development

And for a Python module?

Python eco-system:

- numpy/scipy : scientific usage
- scikit-learn : Machine-Learning algorithms
- pydoe, pydoe2 : Design of experiment
- chaospy : Polynomial chaos expansion
- pandas : Data analysis
- SALib : Sensitivity analysis library
- pymc3 : Fit bayesian models
- GPy, GPFlow, GPyTorch, TensorFlow, PyTorch, georges, ...
- statsmodels : Statistical tools
- $\cdots$ many many

Can my module be used by openturns algorithms?

For now, there is (almost) no way to use a Python object within the `OpenTURNS` C++ library, excepting:

- A Distribution;
- a Function;
- An experiment
- A RandomVector
- A FieldFunction
- ...

and use these one as "OpenTURNS classes".
The concept of "OpenTURNS Python module" is not very specific: we get some 'full python' packages such as `oticp`, `otwrap` , `otsklearn`. But these examples are in a way independent as they rely on the "OpenTURNS API" and the API is not able to 'integrate' them. For example :

- Optimization toolbox even if it relies on `openturns` objects;
- Surrogate models - Could not inherit from `MetaModelAlgorithm`
- kernels - Could not inherit from `CovarianceModel`
- ...

No existing (official) template

In comparison with C++, there is no (official) template. Different reasons:

- Such a module might be some development using `openturns` classes (arguments, internal, return);
- Such a development relies on external modules (`scikit-learn, rpy2` for example)
- Such a development makes others benefit from `openturns` in a friendly framework (`scikit-learn` for example)
- ...

However a template might be used and is now under consortium: new templates
We can have a look at its documentation

How to proceed?

Identify the needs

- Why do we need an `OpenTURNS` module?
- Do I need to link to an (other) external module? If yes what kind of data or objects is required?
- What kind of interaction with the API?
- What does my module require and/or provide?
- Does something similar exists in the API (see `Distribution, Experiment ... for example`? another module?
- ...

Module development

Step 1: copy and adapt an existing template (even if non official) or start a new one

- Copy and rename the source tree of the template. This last one is located hereafter:
  `git clone https://github.com/openturns/ottemplatepython.git MyModule`
- Set the url of the remote (meaning you already created a repository):
  `git remote set-url origin https://github.com/USERNAME/REPOSITORY.git`
- Adapt the template to your module:
  `no customize.sh`
  Rename (manually) folder to the new module, update tests and doc.

Module development

Step 2: develop the module

- Implement your module. You are free to use the rules you want!
- Build & install your module as usual:

```
python setup.py build
python setup.py install
```

Module development

Step 3: documentation

- Document classes and methods using python docstrings.
- Developer guide : Architecture, validation
- Examples and API documentation. Examples using for example notebook converted to scripts or directly use the python sphinx gallery
- Theory (if needed)
- Then build the doc using: `python setup.py build_sphinx`

Module development

Step 4: install and test the module

- Since first step is ok, test your module within python:

  ```
  $python
  >>> import openturns as ot
  >>> import mymodule
  ```

  and python should not complain about a non existing mymodule module.
- You can now use your module as any python module.
- For the setup, use `setuptools, distutils`
- For the tests, rely on `openturns test`, `pytest`

Module development

Step 5: Maintenance
Each developer is responsible of his package should maintain it!

- We can rely on continuous integration tool accounting API changes
- He is in charge of bugs (except if underlying bug is in the API)

Exception : modules maintained by the consortium!

Step 6: Packaging
No specific rule concerning the packaging!

- Tarball generation
- Git
- conda/pip ...
- `python setup.py sdist --formats=gztar,zip`

Exception : modules maintained by the consortium!

Module development: conda deployement

User should generates a recipe in staged-recipes.

- Rely on existing template
- Fork the repository
- Have a look at the documentation
- Adapt to the module and propose pull-request

In case of pure python it is more simple! Have a look at tensap for example.
Once it is done, module is integrated to `conda-forge`

Module development: pip

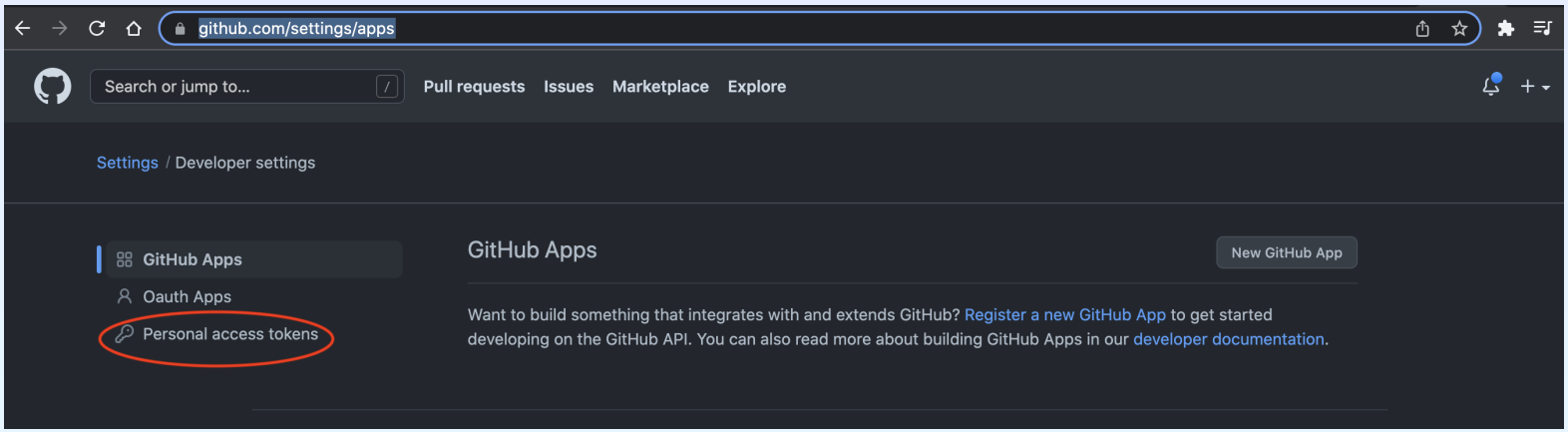
We can rely on this blog to upload the module:

- `pip install twine :  install the module that help the upload`
- `We suppose the package is organized in a well format (Readme, License, module)`
- `python setup.py bdist_wheel to create a .dist`
- `twine upload dist/* to upload.  You should have an account on pypi and enter credential when asked.`

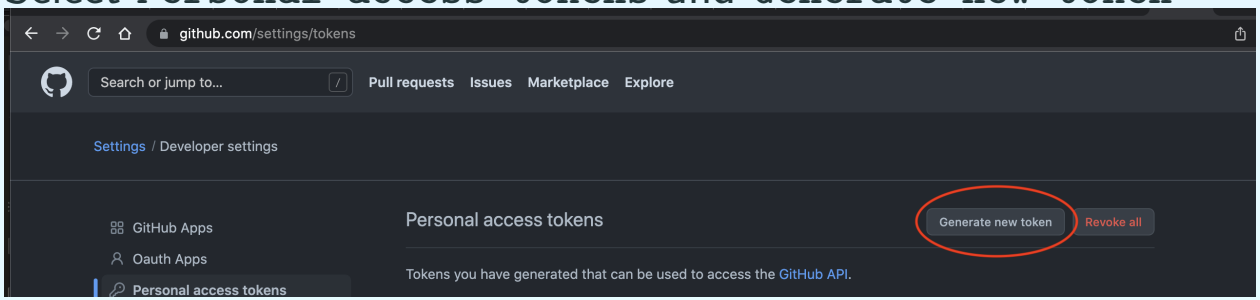
Module development: upload documentation using github and continuous integration

We suppose here that we use github for the maintenance. The purpose here is to illustrate how to deply/upload documentation after a commit on the main branch.

- Goto the `Developer settings` in your github account (see here)

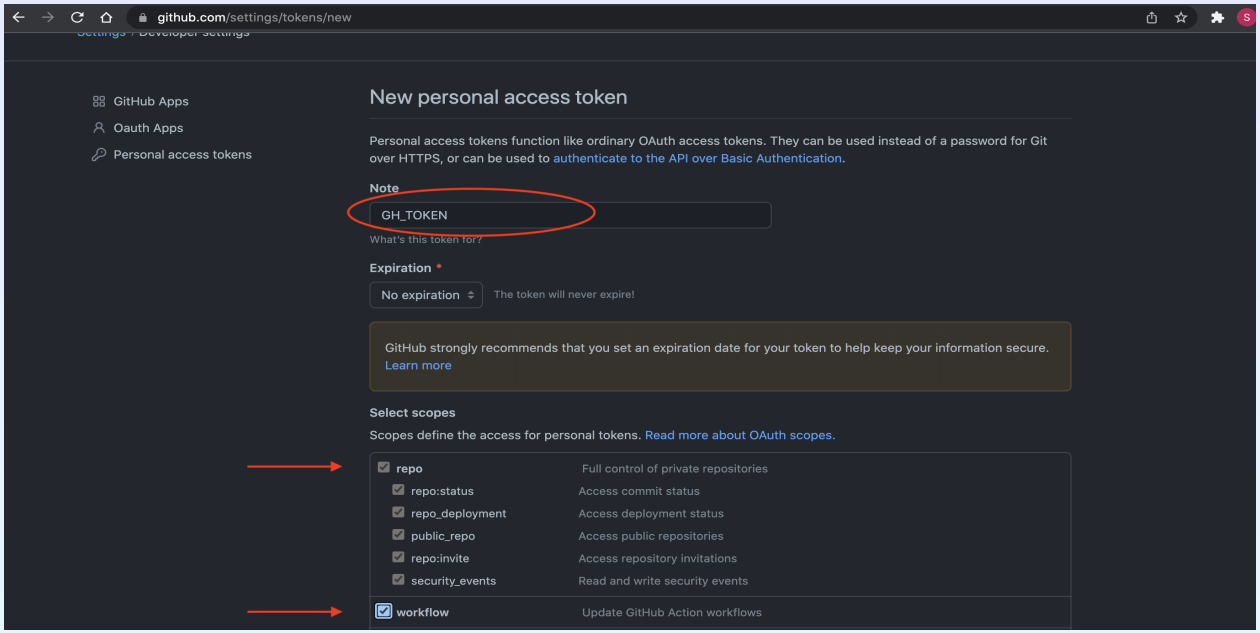


- Select `Personal access tokens` and `Generate new token`

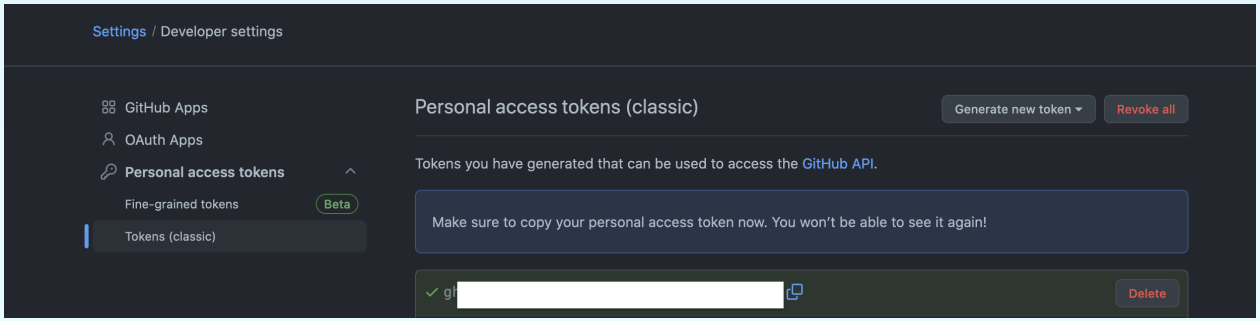


Module development: upload documentation using github and continuous integration

- Provide a name (for example GH_TOKEN) and select at least repo and workflow



- At the end, we generate a token

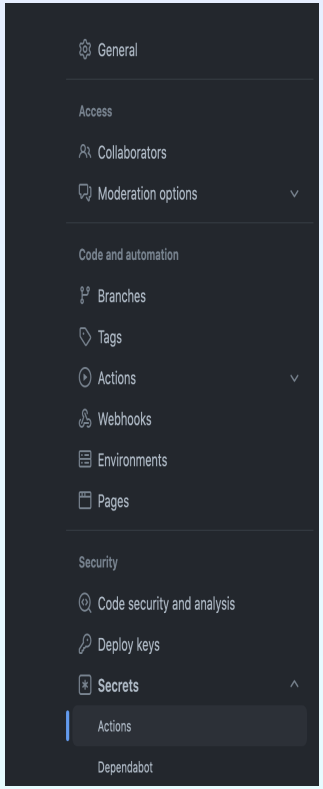


- We should store/copy this token as it will be used later.

Module development: upload documentation using github and continuous integration

Now we should feed corresponding project with the previous token. For that:

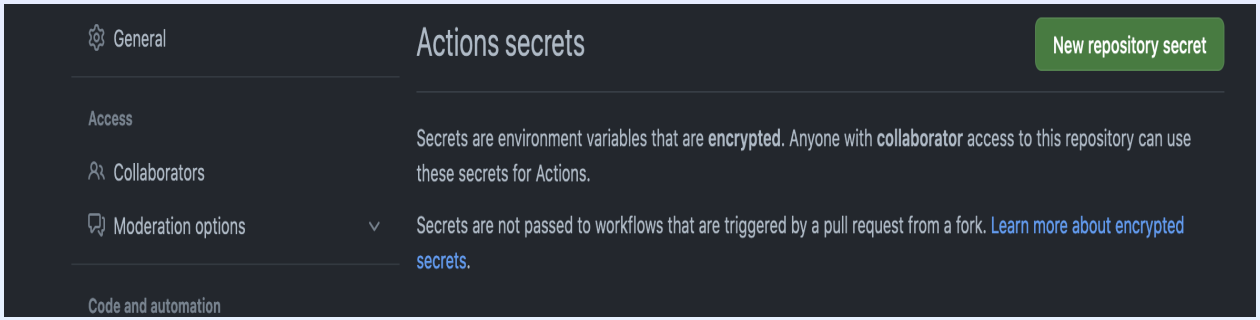
- On github, reach the settings corresponding to the repository of interest.
- On the left, select the `Secret` and `Actions`



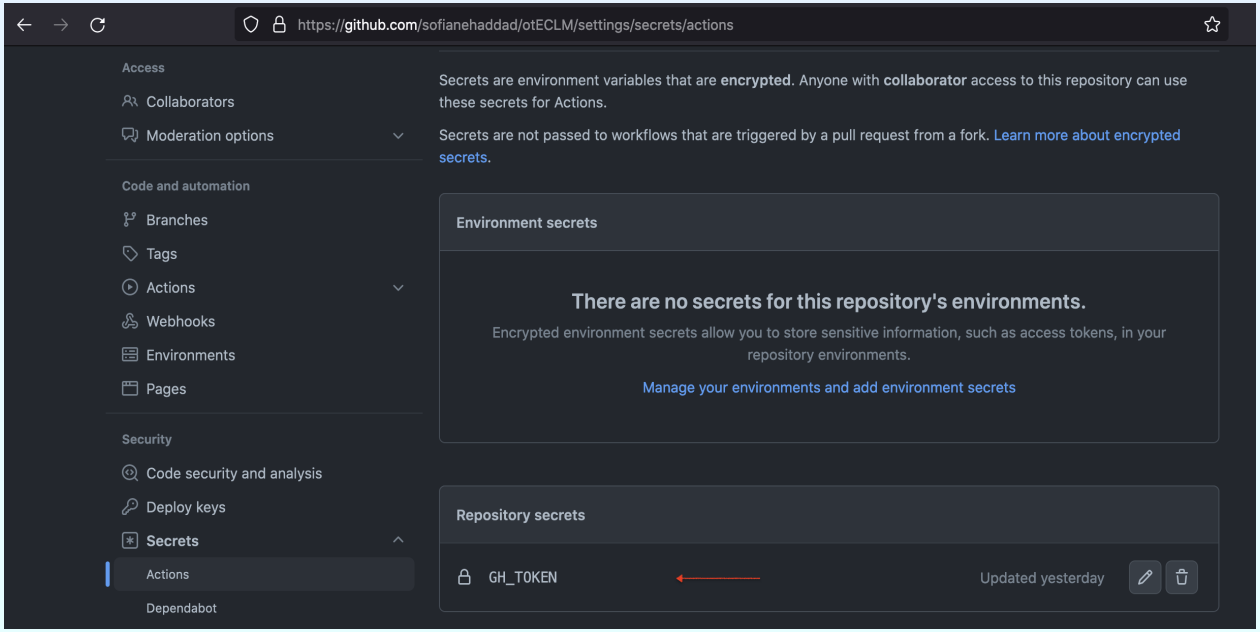
Module development: upload documentation using github and continuous integration

Now we should feed corresponding project with the previous token. For that:

- Generate new repository secret



- Set the name `GH_TOKEN` and paste the previous token generated.
- Now your token is stored and usable within this repository



Module development: upload documentation using github and continuous integration

Finally we have all elements to deploy the documentation. To complete :

- We should make sure to have a repository where upload is done (many often, `username.github.io`).
  This should contains an html index file. A minimal index could be provide.

```
<!-- <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="refresh" content="0; url=http://username.github.io/sphinx/pythonmodule/master" />
</head>
</html>
```

- Add corresponding instructions into continuous-integration script to upload on the previous repository

```
    - name: Upload
      if: ${{ github.ref == 'refs/heads/master' }}
      run: |
        git clone --depth 1 https://${{ secrets.GH_TOKEN }}@github.com/username/username.github.io.git /tmp/io
        mkdir -p /tmp/io/sphinx/pythonmodule/master
        cp -r doc/_build/html/* /tmp/io/sphinx/pythonmodule/master
        cd /tmp/io
        touch .nojekyll
        git config user.email "support@gh.com"
        git config user.name "GitHub Actions"
        git add -A .
        if test `git diff HEAD | wc -c` -eq 0; then exit 0; fi
        git commit -a -m "GitHub Actions build ${GITHUB_REPOSITORY} ${GITHUB_RUN_ID}"
        git push --quiet origin main > /dev/null 2>&1
```

Note: Keep in mind to replace `master` by `main` or other specific branch (release,...) with respect to the branch of interest.