qube Documentation

Release 2.6.1

Lukas Heumos

Dec 04, 2020

CONTENTS:

1 Features 3

2 Credits 5

3 Newbie Guide to QBiC software 7
- 3.1 Required tools for Java/Groovy development 7
- 3.2 What to do once you've generated your project using QUBE? 8

4 Installation 13
- 4.1 Stable release 13
- 4.2 From sources 13

5 General Usage 15
- 5.1 create 15
- 5.2 list 15
- 5.3 info 15
- 5.4 lint 16
- 5.5 bump-version 16
- 5.6 sync 16
- 5.7 config 16
- 5.8 upgrade 16
- 5.9 External Python based projects 17

6 Configure qube 19
- 6.1 qube config all 19
- 6.2 qube config general 19
- 6.3 qube config pat 19
- 6.4 Updating a personal access token 20
- 6.5 Viewing the current configuration 20

7 Create a project 21

8 Getting information about available templates 23
- 8.1 list 23
- 8.2 info 24

9 Linting your project 25
- 9.1 qube linting 25
- 9.2 Linting codes 25

10 Bumping the version of an existing project 29

10.1 Usage . . . 29
10.2 Configuration . . . 31

11 Syncing your project **33**
11.1 Overview . . . 33
11.2 Requirements for sync . . . 33
11.3 How to use sync? . . . 33
11.4 What command line options are available? . . . 34
11.5 What happens when my project gets synced? . . . 34
11.6 Configuring sync . . . 34

12 Upgrade qube **37**

13 Github Support **39**
13.1 Overview . . . 39
13.2 Branches . . . 39
13.3 Github Actions . . . 40
13.4 Secrets . . . 40
13.5 Issue labels . . . 41

14 Available templates **43**
14.1 cli-java . . . 43
14.2 gui-java . . . 45
14.3 lib-java . . . 47
14.4 lib-groovy . . . 49
14.5 service-java . . . 51
14.6 portlet-groovy . . . 53
14.7 portlet-groovy-osgi . . . 55
14.8 Shared FAQ . . . 57

15 Adding new templates **59**
15.1 Template requirements . . . 59
15.2 Step by step guide to adding new templates . . . 60

16 Troubleshooting **65**

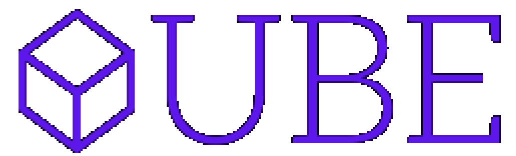
17 Changelog **67**
17.1 2.6.1 (2020-11-06) . . . 67
17.2 2.6.0 (2020-10-27) . . . 67
17.3 2.5.1 (2020-10-16) . . . 67
17.4 2.5.0 (2020-10-06) . . . 68
17.5 2.4.6 (2020-10-02) . . . 68
17.6 2.4.5 (2020-10-02) . . . 68
17.7 2.4.4 (2020-10-02) . . . 69
17.8 2.4.3 (2020-10-01) . . . 69
17.9 2.4.2 (2020-10-01) . . . 69
17.10 2.4.1 (2020-10-01) . . . 69
17.11 2.4.0 (2020-10-01) . . . 70
17.12 2.3.0 (2020-09-28) . . . 70
17.13 2.2.0 (2020-08-21) . . . 70
17.14 2.1.0 (2020-08-21) . . . 71
17.15 2.0.0 (2020-08-17) . . . 71
17.16 1.4.1 (2020-05-23) . . . 72
17.17 1.4.0 (2020-05-23) . . . 72
17.18 1.3.2 (2020-05-22) . . . 72

17.19 1.3.1 (2020-05-20) 72
17.20 1.3.0 (2020-05-20) 73
17.21 1.2.1 (2020-05-03) 73
17.22 1.2.0 (2020-05-03) 73
17.23 1.1.0 (2020-05-03) 74
17.24 1.0.0 (2020-05-03) 74

18 Credits **75**
18.1 Development Lead 75
18.2 Contributors 75

19 Contributor Covenant Code of Conduct **77**
19.1 Our Pledge 77
19.2 Our Standards 77
19.3 Our Responsibilities 77
19.4 Scope 78
19.5 Enforcement 78
19.6 Attribution 78

20 Indices and tables **79**



QBiC's internal project template collection.

- Free software: MIT
- Documentation: https://qube.readthedocs.io.

CHAPTER

ONE

FEATURES

- Create one of QBiC's internal project templates (Java, Groovy or R based)
- List all available templates
- Lint the project to verify that it adheres to QBiC's standards
- Conveniently bump the version of any qube project

CHAPTER

TWO

CREDITS

This package was created with cookietemple based on a modified audreyr/cookiecutter-pypackage project template using Cookiecutter.

CHAPTER

THREE

NEWBIE GUIDE TO QBIC SOFTWARE

This is the newbie guide to developing software for QBiC. This is not solely the documentation for QUBE. If you are looking solely for QUBE's documentation, please proceed with the next sections.

3.1 Required tools for Java/Groovy development

3.1.1 1. Java

Our production and test instances use OpenJDK 1.8 and this is mirrored in our build system.

Installation of OpenJDK varies across operating systems. So it is strongly advised to read more about how to install OpenJDK in your operating system. Most Linux-based operating systems offer OpenJDK through package managers (e.g., `pacman`, `apt`, `yum`).

Make sure you install a Java development kit (JDK) and not just a Java runtime environment (JRE).

3.1.2 2. Maven

Apache Maven is one of those rare tools whose true purpose might be hard to grasp in the beginning, yet it is extremely easy to install. If this is your first time using maven, make sure to read Maven in 5 minutes and Maven getting started guide.

It is up to you to decide whether to install maven using your favorite package manager or install it manually. In any case, make sure you install the most recent version available.

3.1.3 3. Other tools

We use the Travis CI client to *generate encrypted credentials*. You can follow this guide to get the Travis CI client installed on your machine.

3.2 What to do once you've generated your project using QUBE?

QUBE creates just a sample project. Sadly, you will still have to write your own code, tests and documentation.

3.2.1 Write tests, check code coverage

The generated folder already contains simple jUnit 4 unit tests (i.e., in `src/test/java/life/qbic/portal/portlet/DonutPortletTest.java`). Writing code that tests your code is an important part of the development lifecycle (see: https://makeameme.org/meme/Yo-dawg-I-wgn8jg).

As a general guideline, try to code the *logic* of your portlet independent of the user interface so you can easily write code that tests your portlet.

Maven has been configured to execute unit tests under the `src/test` folder that match the `*Test` name (e.g., DonutPortletTest). To run all unit tests, you use the following command:

```
mvn test
```

We use Cobertura to generate coverage reports. To run the unit tests and generate a code coverage report, simply run:

```
mvn cobertura:cobertura
```

Similarly, we have configured the Maven plug-ins to run integration tests. These tests are also under the `src/test` folder, but their names must end with `*IntegrationTest`, such as `DonutPortletIntegrationTest`. Running integration tests can be a time-consuming task, so these are, usually, not executed alongside the unit tests. To execute the integration tests, invoke the following command:

```
mvn verify
```

3.2.2 Test your code locally

You can easily run the unit and integration tests for libraries you have written by using the `mvn test` command. This is, more or less, what our build system does. Take a look at the `.travis.yml` file located in the `common-files` if you want to know all implementation details related to how we do continuous integration.

Testing a portlet locally using Jetty

Go to the generated folder (i.e., `generated/donut-portlet` in our case) and run:

```
mvn jetty:run
```

You should see an output similar to:

```
[INFO] Started ServerConnector@67c06a9e{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
[INFO] Started @30116ms
[INFO] Started Jetty Server
```

Direct your browser to `localhost:8080`. If everything went fine, you will see a portlet with several controls. So far so good, congratulations!

Interact with the UI and, if this is your first portlet, we strongly suggest you to try to change a few things in the code and see what happens after you test again.

3.2.3 Testing other tools locally

We configured a Maven plug-in to generate *stand-alone* JAR files for projects of type cli, service and gui. Maven will package all of the needed dependencies inside one single JAR file.

To test your CLI tool locally, you first need to *package* your artifact using Maven in the generated project folder:

```
mvn package
```

You then need to use the following command:

```
java -jar target/<project_slug>-<version>-jar-with-dependencies.jar
```

That is:

```
java -jar target/donut-cli-1.0.0-SNAPSHOT-jar-with-dependencies.jar
```

3.2.4 Create a new GitHub repository for your new project

You now have a new QBiC project with all the required dependencies and configuration files. You still need to create a remote repository for it, though, so it's available for everyone. QUBE should have prompted you for your Github username and personal access token to conduct this process automatically. If you declined the automatic creation of the repository, then please follow Create Github Repository to create a repository on GitHub. For this example, we will still use `donut-portlet` as the name of our repository. You need to create your GitHub repository under the QBiC GitHub organization, so you need writing access. Ask your favorite QBiC admin if you do not yet have writing rights.

3.2.5 Secure your configuration files before pushing to Git

It might happen that you accidentally pushed a file containing sensitive data. Well - happens.

The good part is that this is reversible. The bad part is that, due to compliance with EU law, whenever one of these incidents occurs, the only way to do this right is to not only to delete all compromised files from the repository, but also to change all compromised passwords, which is a great way to ruin someone's day.

So don't do it, but if you do, or if you discover such an incident, just know that this **should** be reported.

3.2.6 Check that everything worked in Travis-CI.com

The generated project folder contains a `.travis.yml` file that will help you integrate your GitHub repository with Travis CI, our continuous integration service. Broadly speaking, everytime you *push* a change into your GitHub repository, Travis CI will use the `.travis.yml` file to know what to do.

Your repository should have been automatically added to our continuous integration system, but there has been a lot of changes in the platform that your experience might differ. Follow these steps to check that everything worked as advertised:

1. Navigate to (https://travis-ci.com/). Use your GitHub account to authenticate.
2. Click on your name (upper-right corner). You should see your profile in Travis CI.
3. Click the *Sync account* button
4. Look for your repository. You might want to filter repositories by entering the full name of your repository (i.e., `donut-portlet`) or parts of it.

5. Once you've found your repository, click on the *Settings* button displayed next to it.

If you see the settings page, then it means that everything went fine. Make sure that the general settings of your repository match the ones shown below:

3.2.7 Report generation using Maven

We generate reports using the Maven site plugin. The goal `site` will output several reports which you can then find in the `target/site` directory. So, in other words, running `mvn site` will populate the `target/site` folder with all configured reports, as defined in parent-poms.

Reports are generated in HTML format, so you can access them in your browser by entering `file://<full-path-to-your-local-repo>/target/site/index.html` (e.g., `file:///home/homer/donut-portlet/target/site/index.html`; shh... no that's not a typo, that's three forward slashes, remember that full paths start with `/`) in your browser's address bar.

3.2.8 Provide encrypted information to Travis CI

Any person on the internet can download Maven artifacts from our public Maven repository. But in order to upload artifacts to our repository, you will need proper authentication.

Since all of our code is open source, it would not be a good idea to use cleartext passwords and distribute them in our repositories. This is also true for other private information such as license codes. However, Travis CI requires this same information to be present at build time. Luckily, Travis CI offers a simple way to add encrypted environment variables. You do not need to fully understand the implementation details to follow this guide, but no one will be angry at you if you do.

You only need to execute a single command using the Travis CI client to add an encrypted variable to your `.travis.yml`. Let's say, for instance, that you need to add an environment variable, `NUCLEAR_REACTOR_RELEASE_CODE` whose value is `d0nut5_Ar3_t4sty`. You would have to use the following command:

```
travis encrypt "NUCLEAR_REACTOR_RELEASE_CODE=d0nut5_Ar3_t4sty" --add env.global --pro
```

This command will automatically edit `.travis.yml` (if you want edit the file yourself, do not use the `--add env.global` parameter).

3.2.9 Maven credentials

To enable Maven deployments in Travis CI, add both `MAVEN_REPO_USERNAME` and `MAVEN_REPO_PASSWORD` as encrypted variables in your `.travis.yml` file like so:

```
travis encrypt "MAVEN_REPO_USERNAME=<username>" --add env.global --pro
travis encrypt "MAVEN_REPO_PASSWORD=<password>" --add env.global --pro
```

Ask the people who wrote this guide about the proper values of `<username>` and `<password>`. Encrypted values in Travis CI are bound to their GitHub repository, so you cannot simply copy them from other repositories.

3.2.10 Using Vaadin Charts add-on in your portlet

This add-on requires you to have a proper license code. If your portlet requires this add-on, add the `VADIN_CHARTS_LICENSE_CODE` as an encrypted variable in `.travis.yml`:

```
travis encrypt "VAADIN_CHARTS_LICENSE_CODE=<license-code>" --add env.global --pro
```

Ask around for the license code.

3.2.11 Publish your first version

In your local GitHub repository directory (i.e., `donut-portlet`) run the following commands:

```
git init
git add .
git commit -m "Initial commit before pressing the 'flush radioactive material' button"
git remote add origin https://github.com/qbicsoftware/donut-portlet
git push origin master
git checkout -b development
git push origin development
```

Of course, you must replace `donut-portlet` with the real name of your repository. You can now start using your repository containing your brand new portlet.

3.2.12 Change default branch

We strongly recommend you to set the `development` branch as your default branch by following setting the default branch.

3.2.13 Dependabot

Almost all of QUBE's template use Dependabot to automatically submit pull requests to the project's repository whenever an update for a dependency was released. If you pushed your project to Github using QUBE's Github support everything is already setup for you. If not, you may need to create a `development` branch and a `dependabot` issue label.

3.2.14 Read the docs

All of QUBE's templates come with Read the Docs preconfigured.
The only thing left for you to do is to enable your repository for Read the Docs. Please follow the importing your documentation guide. You should not need to manually import your project if you are a member of QBiC software (which you should be!).

3.2.15 Github Actions

All of QUBE's templates feature Github Actions support. Github Actions is part of our continuous integration setup and various template specific workflows are active (on push). Examples are the automatial runs of test suites, package building, linting and more. You usually should not need to touch them, but feel free to add additional workflows. They are located in `.github/workflows/`.

CHAPTER

FOUR

INSTALLATION

4.1 Stable release

To install qube, run this command in your terminal:

```
$ pip install qube
```

This is the preferred method to install qube, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

4.2 From sources

The sources for qube can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/qbicsoftware/qube
```

Or download the tarball:

```
$ curl -OJL https://github.com/qbicsoftware/qube/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

FIVE

GENERAL USAGE

In the following an overview of qube's main commands is given. Please note that all commands are explained more in depth in their respective documentation point. You can use the menu on the left to navigate to them.

5.1 create

`create` is the heart of qube. It starts the project creation process and guides the user through domain selection, language selection and prompts for all required configuration parameters such as name, email and many more. Additionally, the project is linted after creation to ensure that everything went well. The user also has the option to push his just created project directly to Github. Invoke `create` by running

```
$ qube create
```

For more details about project creation please visit *Create a project* and for a detailed list of all available templates please visit *Available templates*.

5.2 list

`list` allows you to list all available templates. The list command prints the name, handle, short description, available libraries for the template and its version to the console. Note that the long description is emitted and the `info` command should be used to get a long description of the template. Invoke `list` by running

```
$ qube list
```

For more details please visit *Getting information about available templates*.

5.3 info

`info` provides detailed information about a specific template or set of templates. It prints the name, handle, long description, available libraries and version of the selected subset or specific template. Invoke `info` by running

```
$ qube info <HANDLE>
```

For more details please visit *Getting information about available templates*.

5.4 lint

`lint` ensures that the template adheres to qube's standards. When linting an already existing project several general checks, which all templates share are performed and afterwards template specific linting functions are run. All results are collected and printed to the user. If any of the checks fail linting terminates. Invoke `lint` by running

```
$ qube lint
```

For more details please visit *Linting your project*.

5.5 bump-version

`bump-version` conveniently bumps the version of a qube based project across several files. Default configurations for `bump-version` are shipped with the template and can be extended if the user so desires. All lines where the version was changed are printed to the console. Invoke `bump-version` by running

```
$ qube bump-version <NEWVERSION> <PATH>
```

For more details please visit *Bumping the version of an existing project*.

5.6 sync

`sync` checks for a project whether a newer version of the used template is available. If so, a pull request with only the changes of the newer template version is created against the development/last active branchh. Invoke `sync` by running

```
$ qube sync
```

For more details please visit *Syncing your project*.

5.7 config

`config` sets commonly used defaults for the project creation. Moreover, it is required for qube's Github support, since it takes care of the personal access token (PAT). Invoke `config` by running

```
$ qube config <all/general/pat>
```

For more details please visit *Configure qube* and *Github Support*.

5.8 upgrade

`upgrade` checks whether a new version is available on PyPI and upgrades the version if not. Invoke `upgrade` by running

```
$ qube upgrade
```

For more details please visit *Upgrade qube*.

5.9 External Python based projects

To use qube in an external Python based project

```
import qube
```

The main functions that you might be interested in can be found here in our repository.

CHAPTER

SIX

CONFIGURE QUBE

To prevent frequent prompts for information, which rarely or never changes at all such as the full name, email or Github name of the user qube uses a configuration file. Moreover, the personal access token associated with the Github username is stored encrypted to increase the security even if malicious attackers already have access to your personal computer. The creation of projects with qube requires a configuration file. A personal access token is not required, if Github support is not used. The configuration file is saved operating system dependent in the usual config file locations (~/.config/qube on Unix, C:UsersUsernameAppDataLocalqubeqube).

Invoke qube config *via*

```
$ qube config <all/general/pat>
```

6.1 qube config all

If `qube config all` is called, the options `general` and afterwards `pat` are called.

6.2 qube config general

`qube config general` prompts for the full name, email address and Github username of the user. If you are not using Github simply keep the default value. These attributes are shared by all templates and therefore should only be set once. If you need to update these attributes naturally rerun the command.

6.3 qube config pat

qube's Github support requires access to your Github repositories to create repositories, add issues labels and set branch protection rules. Github manages these access rights through Personal Access Tokens (PAT). If you are using qube's Github support for the first time `qube config pat` will be run and you will be prompted for your Github PAT. Please refer to the official documentation on how to create one. qube only requires `repo` access, so you only need to tick this box. However, if you want to use qube's sync feature on organisation repositories, you also need to tick `admin:org`. This ensures that your PAT would not even allow for the deletion of repositories. qube then encrypts the Personal Access Token, adds the encrypted token to the `qube_conf.cfg` file (OS dependent stored) and saves the key locally in a hidden place. This is safer than Github's official way, which recommends the usage of environment variables or Github Credentials, which both save the token in plaintext. It is still strongly adviced to secure your personal computer and not allow any foe to get access. If you create a second project using qube at a later stage, you will not be prompted again for your Github username, nor your Personal Access Token.

6.4 Updating a personal access token

If you for any reason need to regenerate your PAT rerun `qube config pat`. Ensure that your Github username still matches. If not you should also update your Github username *via* `qube config general`. Additionally, any of your already created projects may still feature your old PAT and you may therefore run into issues when attempting to push. Hence, you must also update your remote URL for those projects!

6.5 Viewing the current configuration

If you want to see your current configuration (in case of an update for example), you can use `qube config --view`. This will output your current qube configuration. Note that this will only inform you whether your personal access token (PAT) is set or not. It won't actually print its (encrypted) value.

CHAPTER

SEVEN

CREATE A PROJECT

Creating projects from templates is the heart of qube.

Our templates adhere to best practices and try to be as modern as possible. Furthermore, they try to automate tasks such as automatical dependency resolvement and installation, packaging, deployment and more.

To learn more about our templates please visit *Available templates* and check out your template of interest.

The creation of a new project can be invoked by

```
$ qube create
```

which will guide you through the creation process of your (customized) project via prompts. If you do not have a qube config yet, you will be asked to create one first. The full name, email and possibly more information set during the configuration process is required when creating the project. For more details please visit *Configure qube*. The prompts follow the pattern of domain (e.g. cli, gui, ...), subdomain (if applicable, e.g. website), language (e.g. Python) followed by template specific prompts (e.g. testing frameworks, ...). | The template will be created at the current working directory, where qube has been called.

It is also possible to directly create a specific template using its handle

```
$ qube create --handle <HANDLE>
```

After the template has been created, linting (see *Linting your project*) is automatically performed to verify that the template creation process was successful.

You may already be made aware of any TODOs, which you should examine before coding your project.

Finally, you will be asked whether or not you want to automatically push your new project to Github. For more details about the Github support please visit *Github Support*.

Note that in order to use the automatic Github repo creation feature, you need to set a personal access token (for login, since a login via password will be deprecated in Autumn 2020) via `qube config pat` (if not already done). This token is also used for qube's sync feature.

Take a look at the Github docs to see, how to create a personal access token for your Github account.

CHAPTER EIGHT

GETTING INFORMATION ABOUT AVAILABLE TEMPLATES

Although, information on all QUBE templates is provided in *Available templates* in our documentation, it is often times more convenient to get a quick overview from the commandline.

Hence, QUBE provides two commands `list` and `info`, which print information on all available templates with different levels of detail.

8.1 list

QUBE list can be invoked via:

```
qube list
```

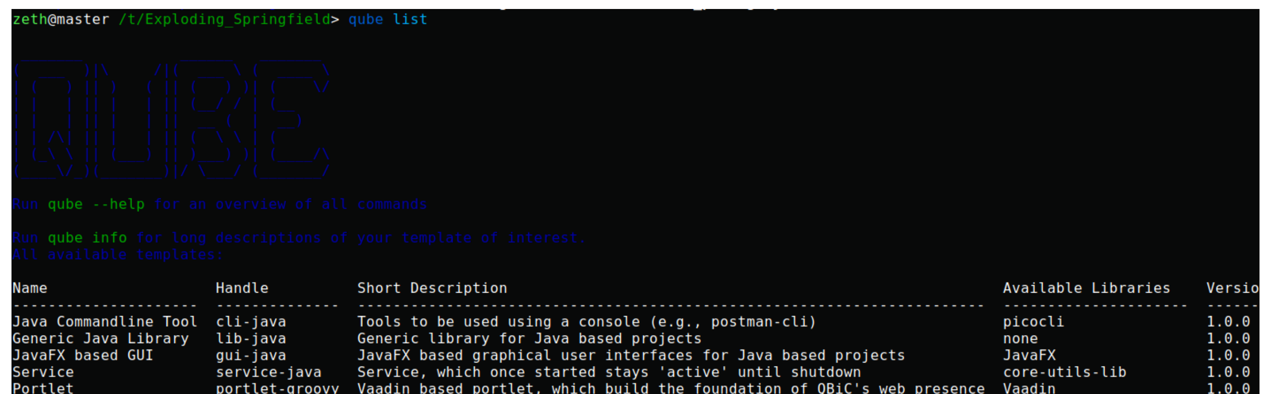


Fig. 1: Example output of `qube list`. Note that the content of the output is of course subject to change.

`qube list` is restricted to the short descriptions of the templates. If you want to read more about a specific (sets of) template, please use the *info* command.

8.2 info

The `info` command should be used when the short description of a template is not sufficient and a more detailed description is required.
Moreover, when you are unsure which template suits you best and you would like to read more about a specific subset of templates further, `info` is your friend.

Invoke `qube info` *via*:

qube info <HANDLE>

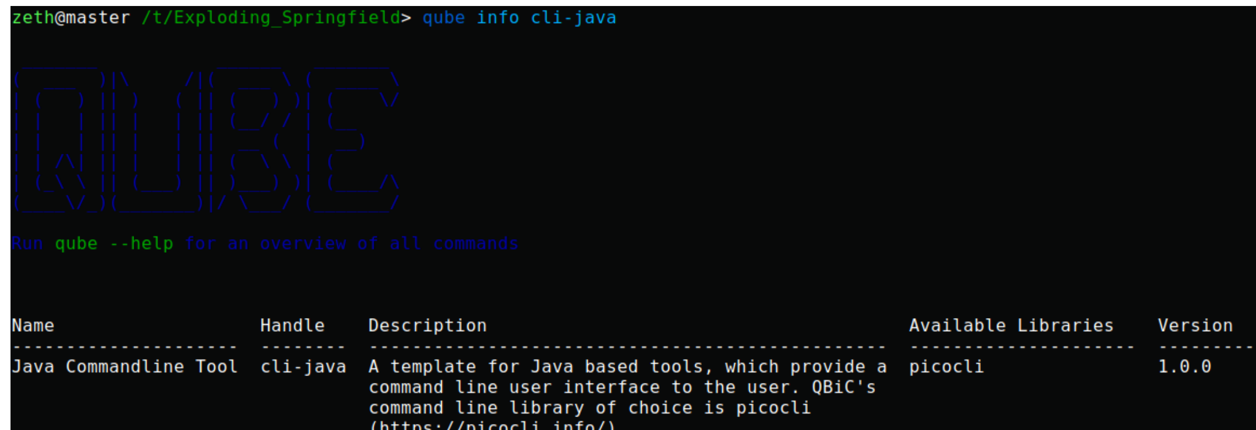


Fig. 2: Example output of `qube info`. The handle can also be shortened to e.g. just *cli*, to output all command line templates of QUBE.

It is not necessary to use a full handle such as *cli-java*. Alternatively, a subset of the handle such as *cli* can be used and as a result detailed information on all templates of the requested domain will be printed.

CHAPTER

NINE

LINTING YOUR PROJECT

Linting is the process of statically analyzing code to find code style violations and to detect errors. qube implements a custom linting system, but depending on the template external tools linting tools may additionally be called.

9.1 qube linting

qube lint can be invoked on an existing project using

```
$ qube lint <OPTIONS> <PATH>
```

qube's linting is divided into three distinct phases.

1. All linting functions, which all templates share are called and the results are collected.
2. Template specific linting functions are invoked and the results are appended to the results of phase 1
3. Template specific external linters are called (e.g. autopep8 for Python based projects)

The linting results of the first two phases are assigned into 3 groups:

1. Passed
2. Passed with warning
3. Failed

If any of the checks failed linting stops and returns an error code.

To examine the reason for a failed linting test please follow the URL. All reasons are explained in the section *Linting codes*.

9.2 Linting codes

The following error numbers correspond to errors found during linting. If you are not sure why a specific linting error has occurred you may find more information using the respective error code.

```
zeth@master /t/Exploding_Springfield> qube lint .

Run qube --help for an overview of all commands

Running general linting
Running portlet-groovy linting
===================================
          LINTING RESULTS
===================================
  [✔]   24 tests passed
  [!]    0 tests had warnings
  [✗]    0 tests failed
Test passed:
 https://cookietemple/linting/errors#1 : File found: qube.cfg
  https://cookietemple/linting/errors#1 : File found: .qube.yml
  https://cookietemple/linting/errors#1 : File found: CODEOFCONDUCT.rst
  https://cookietemple/linting/errors#1 : File found: README.rst
  https://cookietemple/linting/errors#1 : File found: CHANGELOG.rst
  https://cookietemple/linting/errors#1 : File found: LICENSE or LICENSE.md or LICENCE or LICENCE.md
  https://cookietemple/linting/errors#1 : File found: docs/index.rst
  https://cookietemple/linting/errors#1 : File found: docs/readme.rst
  https://cookietemple/linting/errors#1 : File found: docs/changelog.rst
  https://cookietemple/linting/errors#1 : File found: docs/installation.rst
  https://cookietemple/linting/errors#1 : File found: docs/usage.rst
  https://cookietemple/linting/errors#1 : File found: .travis.yml
  https://cookietemple/linting/errors#1 : File found: .travis.settings.xml
  https://cookietemple/linting/errors#1 : File found: .gitignore
  https://cookietemple/linting/errors#1 : File found: .dependabot/config.yml
  https://cookietemple/linting/errors#1 : File found: .github/workflows/build_docs.yml
  https://cookietemple/linting/errors#1 : File found: .github/ISSUE_TEMPLATE/bug_report.md
  https://cookietemple/linting/errors#1 : File found: .github/ISSUE_TEMPLATE/feature_request.md
  https://cookietemple/linting/errors#1 : File found: .github/ISSUE_TEMPLATE/general_question.md
  https://cookietemple/linting/errors#1 : File found: .github/pull_request_template.md
  https://cookietemple/linting/errors#5 : Versions were consistent over all files
  https://cookietemple/linting/errors#1 : File found: pom.xml
  https://cookietemple/linting/errors#1 : File found: .github/workflows/build_docs.yml
  https://cookietemple/linting/errors#1 : File found: .github/workflows/build_package.yml
```

Fig. 1: Linting applied to a newly created cli-java project.

9.2.1 General

general-1

File not found. This error occurs when your project does not include all of qube's files, which all templates share. Please create the file and populate it with appropriate values. You should also critically reflect why it is missing, since at the time of the project creation using qube this file should not have been missing!

general-2

Dockerfile invalid. This error usually originates from empty Dockerfiles or missing FROM statements.

general-3

TODO String found. The origin of this error are `QUBE TODO` strings in the respective files. Usually, they point to things that should be manually configured or require other attention. You may remove them if there is no task for you to be solved.

general-4

Cookiecutter String found. This error occurs if something went wrong at the project creation stage. After a project has been created using qube there should not be any jinja2 syntax statements left. Web development templates may pose exceptions. However, `{{ *cookiecutter* }}` statements should definitely not be present anymore.

general-5

Versions not consistent. If the version of all files specified in the [bumpversion] sections defined in the qube.cfg file are not consistent, this error may be found. Please ensure that the version is consistent! If you need to exclude specific lines from this check please consult *Bumping the version of an existing project*. To prevent this error you should only increase the version of your project using `qube bump-version`.

general-6

`changelog.rst` invalid. The `changelog.rst` file requires that every changelog section has a header with the version and the corresponding release date. The version above another changelog section should always be *greater* than the section below (e.g. 1.1.0 above 1.0.0). Every section must have the headings `**Added**`, `**Fixed**`, `**Dependencies**` and `**Deprecated**`.

CHAPTER

TEN

BUMPING THE VERSION OF AN EXISTING PROJECT

Increasing the version of an already existing project is often times a cumbersome and error prone process, since the version has to be changed in multiple places.

To facilitate this process, qube provides a `bump-version` command, which conveniently increases the version across several files.

Additionally, always adding new sections to the changelog is an annoying process. bump-version therefore inserts a new section into the changelog using the specified new version.

```
New Version (Date)
------------------

**Added**

**Fixed**

**Dependencies**

**Deprecated**
```

bump-version will verify that your new version adheres to semantic versioning and that you are not trying to update it unreasonably. It is for example not allowed to bump from 2.0.0 to 7.1.2, since in a normal development workflow only 2.0.1, 2.1.0 or 3.0.0 adhere to consecutive semantic versioning. Note that SNAPSHOT versions are allowed! Hence, `qube bump-version 1.2.5-SNAPSHOT` is allowed. However, it must still follow semantic versioning. Version 1.2.5 therefore cannot be the predecessor of 1.2.5-SNAPSHOT, but only 1.2.4.

10.1 Usage

The `bump-version` command follows the syntax

```
$ qube bump-version <OPTIONS> X.X.X <PATH>
```

where `X` corresponds to a (python)-integer value of any possible range.

The PATH corresponds to the path to the qube.cfg file, which contains all locations, where the version should be increased.

Note that you don´t need to specify a path, if your current working directory contains the `qube.cfg` file.

Use the `--downgrade` option to downgrade your version. The changelog will not be changed. Only use this option

```
zeth@master /t/Exploding_Springfield> qube bump-version 1.1.0

Run qube --help for an overview of all commands

Changing version number.
Current version is 1.0.0.
New version will be 1.1.0

Updating version number in ./.qube.yml
- version: 1.0.0-SNAPSHOT
+ version: 1.1.0-SNAPSHOT

Updating version number in ./docs/conf.py
- version = '1.0.0'
+ version = '1.1.0'

- release = '1.0.0'
+ release = '1.1.0'

Updating version number in ./pom.xml
- <version>1.0.0-SNAPSHOT</version>
+ <version>1.1.0-SNAPSHOT</version>
```

Fig. 1: bump-version applied to a fresh cli-python project

as a last resort if something went horribly wrong in your development process. In a normal development workflow this should never be necessary.

10.2 Configuration

All templates of qube ship with a `qube.cfg` file, which defines all files `bump-version` examines.

The `bump-version` configuration begins with the section:

```
[bumpversion]
current_version = 0.1.0
```

where the current version is defined. All files are either white- or blacklisted (see below for explanations). An arbitrary name is followed by the path to the file: `arbitrary_name = path_to_file`.

Whitelisted files are listed below a `[bumpversion_files_whitelisted]` section, e.g.:

```
[bumpversion_files_whitelisted]
dot_qube = .qube.yml
conf_py = docs/conf.py
```

All files, which are whitelisted are searched for patterns matching `X.X.X`, which are updated to the specified new versions.
Any lines, which contain the string `<<QUBE_NO_BUMP>>` will be ignored.

If files like Maven pom.xml files, contain many version patterns matching `X.X.X`, it may be a better idea to blacklist them (section `[bumpversion_files_blacklisted]`) and enable only specific lines to be updated:

```
[bumpversion_files_blacklisted]
pom = pom.xml
```

Analogously to whitelisted files, which allow for specific lines to be ignored, blacklisted files allow for specific lines to be forcibly updated using the string `<<QUBE_FORCE_BUMP>>`.

Note that those tags must be on the same line as the version (commonly placed in a comment), otherwise they wont work!

CHAPTER

ELEVEN

SYNCING YOUR PROJECT

Syncing is supposed to integrate any changes to the qube templates back into your already existing project.

11.1 Overview

When `qube sync` is invoked, qube checks whether a new version of the corresponding template for the current project is available. If so, qube now creates a temporary project with the most recent template and pushes it to the `TEMPLATE` branch. Next, a pull request is submitted to the `development` or most recently used branch. The syncing process is configurable by setting the desired lower syncing boundary level and blacklisting files from syncing (see *Sync level*.

11.2 Requirements for sync

For syncing to work properly, your project has to satisfy a few things:

1.) A Github repository with your projects code (private or public, organization or non-organization repository).

2.) An unmodified `.qube.yml` file (if you modified this file, what you should never do, syncing may not be able to recreate the project with the most recent template.

3.) A running, unmodified workflow called `sync.yml`. Modifying this workflow should never be done and results in undefined sync behaviour.

4.) An active repository secret called `QUBE_SYNC_TOKEN` for your project's repository containing the encrypted personal access token with at least `repo` scope.

5.) It is strongly advised not to touch the `TEMPLATE` branch.

11.3 How to use sync?

The `sync` command from qube has a few options to run with. For a first overview see `$ qube sync --help`. Note that you never need to run `qube sync` manually with the workflow, but you can do any time if you want.

11.4 What command line options are available?

The basic sync command syntax is: `$ qube sync [PROJECT_DIR] --set-token ([PAT] [GITHUB_USERNAME]) --check-update`

Running sync manually on an active qube project with a Github repo, you should never ever have to set the `PAT` or `GITHUB_USERNAME`. These are options that are only required for the `sync_project` workflow for automatic sycning. So: **You never need to care about these parameters when calling qube sync manually**.

An important parameter is `PROJECT_DIR`. This parameter contains the (relative) path to the qube projects top level directory, you would like to sync. Per default, this one is set to the current working directory. So, for example, if your current working directory is like `/home/homersimpson/projects` and you would like to sync your project named `ExplodingSpringfield` located in `/home/homersimpson/projects` you need to call `$ qube sync ExplodingSpringfield/`. This one should be always set (unless your current working directory is the top level directory of the project you'd like to sync).

Next, `sync` provides two flags: `$ qube sync [PROJECT_DIR] --set-token` can be used to update your `QUBE_SYNC_TOKEN`, which qube uses to sync your project (especially when syncing with the workflow). This could be useful, for example, when the ownership of a repo had changed.

The `--check_update` flag, called via `$ qube sync [PROJECT_DIR] --check-update`, can be used for manually checking whether a new version for your template has been released by qube. Note that when you call `$ qube sync [PROJECT_DIR]` qube also runs this check, but then proceeds with syncing rather than exiting.

11.5 What happens when my project gets synced?

Syncing can happen via two ways: One way is when you call `$ qube sync [PROJECT_DIR]` manually from your command line. This way, qube checks whether a new version has been released or not, and if so, creates a pull request with all changes (excluding blacklisted files) from the `TEMPLATE` branch to your current working branch.

The other way would be via the `sync_project.yml` workflow. This workflow triggers on push everytime you push changes to your repository. You can safely modify this behaviour to only trigger this workflow for example when a PR is created. The result is the same like above but you don't need to remember to run sync manually on a regular basis.

Note that the PR is currently automatically created by the one who initially created/owns this repository.

11.6 Configuring sync

11.6.1 Sync level

Since qube strongly adheres to semantic versioning our templates do too. Hence, it is customizable whether only major, minor or all (=patch level) releases of the template should trigger qube sync. The sync level therefore specifies a lower boundary. It can be configured in the:

```
[sync_level]
ct_sync_level = minor
```

section.

11.6.2 Blacklisting files

Although, qube only submits pull requests for files, which are part of the template sometimes even those files should be ignored. Examples could be any html files, which at some point contain only custom content and should not be synced. When syncing, qube examines the `qube.cfg` file and ignores any file patterns (e.g. `*.html`) below the `[sync_files_blacklisted]` section.

CHAPTER TWELVE

UPGRADE QUBE

Every time qube is run it will automatically contact PyPI to check whether the locally installed version of qube is the latest version available. If not

```
$ qube upgrade
```

can be run. The command calls pip in upgrade mode to upgrade qube to the latest version. For this to work however, it is required that pip is accessible from your PATH.

It is advised not to mix installations using setuptools directly and pip. If you are not a developer of qube this should not concern you.

CHAPTER

THIRTEEN

GITHUB SUPPORT

13.1 Overview

qube uses GitPython and PyGithub to automatically create a repository, add, commit and push all files. Moreover, issue labels, a development and a TEMPLATE branch are created. The TEMPLATE branch is required for *Syncing your project* to work and should not be touched manually.

13.2 Branches

13.2.1 Overview

git branches can be understood as diverging copies of the main line of development and facilitate parallel development. To learn more about branches read Branches in a Nutshell of the Pro Git Book. A simple best practice development workflow follows the pattern that the `master` branch always contains the latest released code. It should only be touched for new releases. Code on the `master` branch must compile and be as bug free as possible. Development takes place on the `development` branch. All parallelly developed features eventually make it into this branch. The `development` branch should always compile, but it may contain incomplete features or known bugs. qube creates a `TEMPLATE` branch, which is required for *Syncing your project* to work and should not be touched manually.

13.2.2 Branch protection rules

qube sets several branch protection rules, which enforce a minimum standard of best branch practices. For more information please read about protected branches. The following branch protection rules only apply to the `master` branch:

1. Required review for pull requests: A pull request to `master` can only be merged if the code was at least reviewed by one person. If you are developing alone you can merge with your administrator powers.
2. Dismiss stale pull request approvals when new commits are pushed.

13.3 Github Actions

13.3.1 Overview

Modern development tries to merge new features and bug fixes as soon as possible into the `development` branch, since big, diverging branches are more likely to lead to merge conflicts. This practice is known as continuous integration (CI). Continuous integration is usually complemented with automated tests and continuous delivery (CD). All of qube's templates feature Github Actions as main CI/CD service. Please read the Github Actions Overview for more information. On specific conditions (usually push events), the Github Actions workflows are triggered and executed. The developers should ensure that all workflows always pass before merging, since they ensure that the package still builds and all tests are executed successfully.

13.3.2 pr_to_master_from_patch_release_only workflow

All templates feature a workflow called `pr_to_master_from_patch_release_only.yml`. This workflow runs everytime a PR to your projects master branch is created. It fails, if the PR to the `master` branch origins from a branch that does not contain `PATCH` or `release` in its branch name. If development code is written on a branch called `development``and a new release of the project is to be made, one should create a ``release` branch only for this purpose and then merge it into `master` branch. This ensures that new developments can already be merged into `development`, while the release is finally prepared. The `PATCH` branch should be used for required `hotfixes` (checked out directly from `master` branch) because, in the meantime, there might multiple developments going on at `development` branch and you dont want to interfere with them.

13.3.3 sync_project.yml

All templates also feature this workflow. This workflow is used for automatic syncing (if enabled) your project with the latest qube template version. The workflow is run on push events, although this behavior can be customized if desired. The workflow calls `qube sync`, which first checks whether a new template version is available and if so it submits a pull request. For more details please visit *Syncing your project*.

13.4 Secrets

Github secrets are what their name suggests: Encrypted secret values in a repository or an organisation; once they are set their value can be used for sensible data in a project or an organisation but their raw value can never be seen again even by an administrator (but it can be updated).

qube uses a secret called `QUBE_SYNC_TOKEN` for its syncing feature. This secret is automatically created during the repo creation process, if you choose to create a GitHub repo. The secret contains your encrypted personal access token as its value. Note that this will have no effect on how to login or any other activity in your project. If you remove the secret or change its value (even with another personal access token of you) the syncing feature will no longer work. In case you are creating an organisation repository, the secret will also be stored as a repository secret, only usable for your specific project.

See section below in case your Github repository creation failed during the create process.

13.4.1 Error Handling due to failed Github repo creation

Errors during the create process due to a failed Github repo creation may occur due to a vast amount of reasons: Some common error sources are:

1. You have no active internet connection or your firewall protects you against making calls to external APIs.

2. The Github API service or Github itself is unreachable at the moment, which can happen from time to time. In doubt, make sure to check the Github status page.

3. A repo with the same name already exists in your account/your organisation.

Creation fails, ok: But how can I then access the full features of qube? You can try to fix the issue (or wait some time on case, for example, when Github is down) and then process to create a Github repostiryo manually. After this, make sure to create a secret named `QUBE_SYNC_TOKEN` with the value of your PAT for your repository. See the Github docs for more information on how to create a secret.

13.5 Issue labels

qube's Github support automatically creates issue labels. Currently the following labels are automatically created: https://en.wikipedia.org/wiki/Continuous_integration 1. dependabot: All templates, which include Dependabot support label all Dependabot pull requests with this label.

CHAPTER

FOURTEEN

AVAILABLE TEMPLATES

qube currently has the following templates available:

1. *cli-java*
2. *gui-java*
3. *lib-java*
4. *lib-groovy*
5. *service-java*
6. *portlet-groovy*
7. *portlet-groovy-osgi*

In the following every template is devoted its own section, which explains its purpose, design, included frameworks/libraries, usage and frequently asked questions. A set of frequently questions, which all templates share see here: *Shared FAQ* FAQ. It is recommended to use the sidebar to navigate this documentation, since it is very long and cumbersome to scroll through.

14.1 cli-java

14.1.1 Purpose

cli-java is the template of choice for picocli based command line applications, which require a JVM and the related ecosystem (e.g. due to requirement to use OpenBIS).

These applications result in jars and can therefore only be started when a JDK is present. They do not ship with custom JREs!

Therefore, if you are not bound to the JVM ecosystem, it may be a better idea to go for Python based projects, since Python is installed and available on every Unix machine.

QUBE currently does not offer a cli-python template, but you may make use of COOKIETEMPLE's cli-python.

14.1.2 Design

This template follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
│   ├── requirements.txt
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── java_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
├── README.rst
├── .readthedocs.yml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── life
│   │   │       └── qbic
│   │   │           └── cli
│   │   │               ├── SampleCommand.java
│   │   │               ├── SampleEntryPoint.java
│   │   │               └── SampleTool.java
│   │   └── resources
│   │       ├── log4j2.xml
│   │       └── tool.properties
│   └── test
│       ├── java
│       │   └── life
│       │       └── qbic
│       │           └── cli
│       │               ├── SampleIntegrationTest.java
│       │               └── SampleTest.java
│       │
```

(continues on next page)

(continued from previous page)

```
│       └── resources
│           ├── log4j2.xml
│           └── tool.properties
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.1.3 Included frameworks/libraries

1. Like all of QBiC's JVM based projects, cli-java uses QBiC's parent-pom.
2. cli-java uses picocli to expose the commandline parameters to the user.
3. junit4 is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
4. Preconfigured ReadTheDocs.
5. Four Github workflows are shipped with the template
    1. `build_docs.yml`, which builds the ReadTheDocs documentation.
    2. `build_package.yml`, which builds the Maven based project.
    3. `java_checkstyle.yml`, which runs Checkstyle using Google's Styleguides.
    4. `run_test.yml`, which runs all junit4 tests.
    5. `qube_lint.yml`, which runs QUBE's linting on the project.
    6. `pr_to_master_from_development_only.yml` which fails if the PR does not come from a release or hotfix branch

14.1.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-java based projects.

14.2 gui-java

14.2.1 Purpose

gui-java is QBiC's choice for all templates, which require a Desktop user interface. Hence, the application is not necessarily on the web.
The GUI framework of choice is JavaFX 8. Be aware that most of the documentation has already moved past version 8.

14.2.2 Design

gui-java follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
│   ├── requirements.txt
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── java_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
├── README.rst
├── .readthedocs.yml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── life
│   │   │       └── qbic
│   │   │           └── gui
│   │   │               ├── SampleApplication.java
│   │   │               ├── SampleCommand.java
│   │   │               └── SampleEntryPoint.java
│   │   └── resources
│   │       ├── log4j2.xml
│   │       └── tool.properties
│   └── test
│       ├── java
│       │   └── life
│       │       └── qbic
│       │           └── gui
│       │               ├── SampleIntegrationTest.java
│       │               └── SampleTest.java
│       │
```

(continues on next page)

(continued from previous page)

```
│         └── resources
│             ├── log4j2.xml
│             └── tool.properties
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.2.3 Included frameworks/libraries

1. Like all of QBiC's JVM based projects, lib-java uses QBiC's parent-pom.
2. gui-java uses JavaFX 8 to build the graphical user interface.
3. junit4 is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
4. Preconfigured ReadTheDocs.
5. Four Github workflows are shipped with the template
    1. `build_docs.yml`, which builds the ReadTheDocs documentation.
    2. `build_package.yml`, which builds the Maven based project.
    3. `java_checkstyle.yml`, which runs Checkstyle using Google's Styleguides.
    4. `run_test.yml`, which runs all junit4 tests.
    5. `qube_lint.yml`, which runs QUBE's linting on the project.
    6. `pr_to_master_from_development_only.yml` which fails if the PR does not come from a release or hotfix branch

14.2.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-java based projects.

14.3 lib-java

14.3.1 Purpose

cli-lib is by design more open end than the other templates. The goal of a cli-lib project is not to be run as a standalone, but rather to be included as a library in other JVM based projects.

14.3.2 Design

lib-java follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
│   ├── requirements.txt
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── java_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
├── README.rst
├── .readthedocs.yml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── life
│   │   │       └── qbic
│   │   │           └── package-info.java
│   │   └── resources
│   │       └── .gitignore
│   └── test
│       ├── java
│       │   └── life
│       │       └── qbic
│       │           └── package-info.java
│       └── resources
│           └── log4j2.xml
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.3.3 Included frameworks/libraries

1. Like all of QBiC's JVM based projects, lib-java uses QBiC's parent-pom.
2. junit4 is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
3. Preconfigured ReadTheDocs.
4. Four Github workflows are shipped with the template
    1. `build_docs.yml`, which builds the ReadTheDocs documentation.
    2. `build_package.yml`, which builds the Maven based project.
    3. `java_checkstyle.yml`, which runs Checkstyle using Google's Styleguides.
    4. `run_test.yml`, which runs all junit4 tests.
    5. `qube_lint.yml`, which runs QUBE's linting on the project.
    6. `pr_to_master_from_development_only.yml` which fails if the PR does not come from a release or hotfix branch

14.3.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-java based projects.

14.4 lib-groovy

14.4.1 Purpose

lib-groovy is by design more open end than the other templates. The goal of a cli-lib project is not to be run as a standalone, but rather to be included as a library in other JVM based projects.

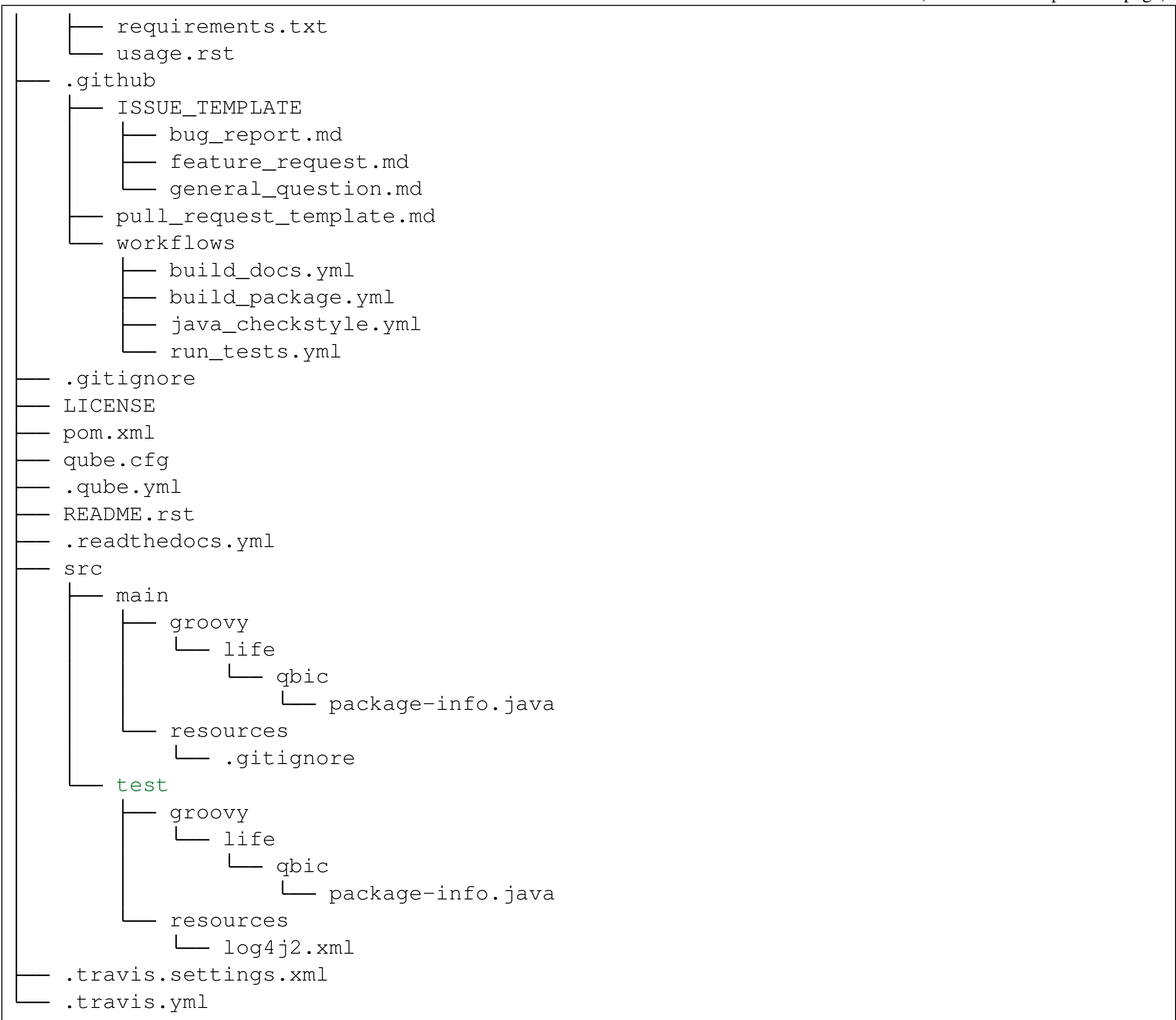
14.4.2 Design

lib-groovy follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
```

(continues on next page)

(continued from previous page)



```
│   ├── requirements.txt
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── java_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
├── README.rst
├── .readthedocs.yml
├── src
│   ├── main
│   │   ├── groovy
│   │   │   └── life
│   │   │       └── qbic
│   │   │           └── package-info.java
│   │   └── resources
│   │       └── .gitignore
│   └── test
│       ├── groovy
│       │   └── life
│       │       └── qbic
│       │           └── package-info.java
│       └── resources
│           └── log4j2.xml
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.4.3 Included frameworks/libraries

1. Unlike all of QBiC's JVM based projects, lib-groovy does not use QBiC's parent-pom for now.
2. spock is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
3. Preconfigured ReadTheDocs.
4. Four Github workflows are shipped with the template
   1. `build_docs.yml`, which builds the ReadTheDocs documentation.
   2. `build_package.yml`, which builds the Maven based project.
   3. `java_checkstyle.yml`, which runs Checkstyle using Google's Styleguides.
   4. `run_test.yml`, which runs all junit4 tests.

5. `qube_lint.yml`, which runs QUBE's linting on the project.
6. `pr_to_master_from_development_only.yml` which fails if the PR does not come from a release or hotfix branch

14.4.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-groovy based projects.

14.5 service-java

14.5.1 Purpose

service-java is a base template for services, which are similar to commandline tools, but stay active until shutdown.

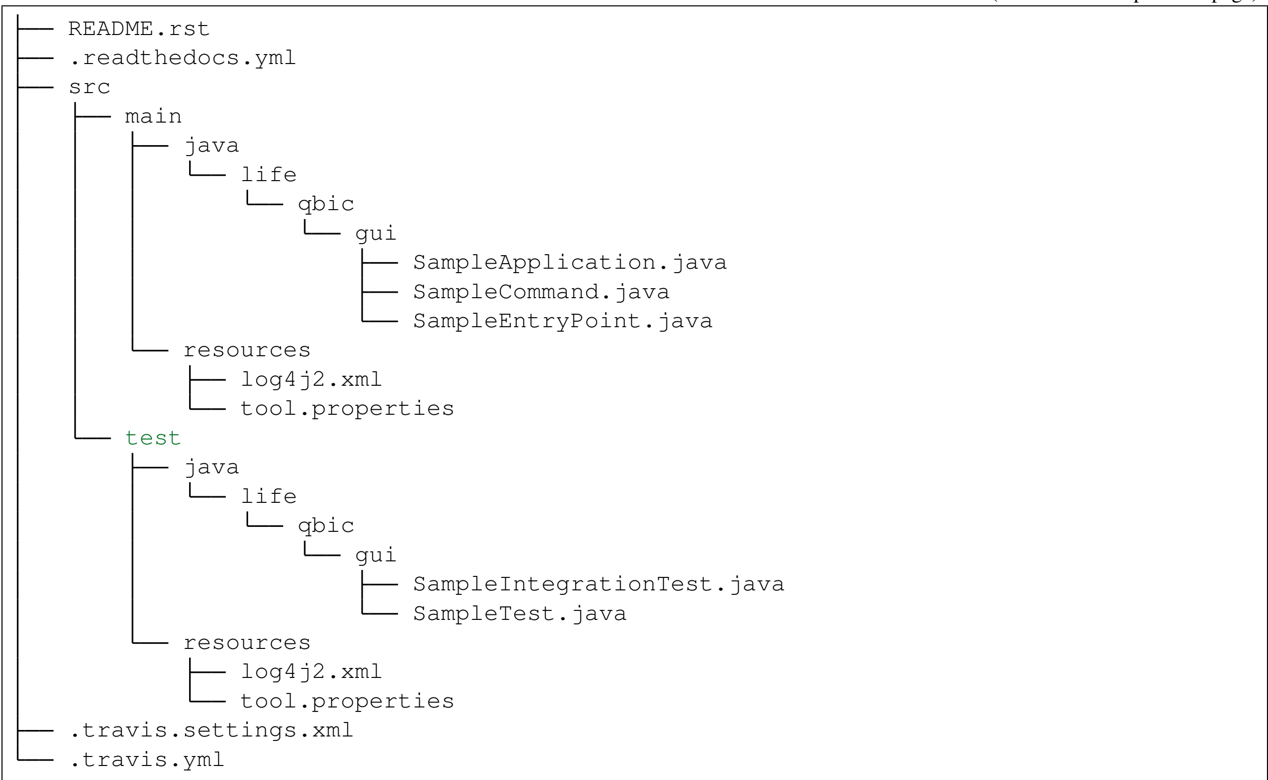
14.5.2 Design

service-java follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
│   ├── requirements.txt
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── java_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
```

(continues on next page)

(continued from previous page)



```
├── README.rst
├── .readthedocs.yml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── life
│   │   │       └── qbic
│   │   │           └── gui
│   │   │               ├── SampleApplication.java
│   │   │               ├── SampleCommand.java
│   │   │               └── SampleEntryPoint.java
│   │   └── resources
│   │       ├── log4j2.xml
│   │       └── tool.properties
│   └── test
│       ├── java
│       │   └── life
│       │       └── qbic
│       │           └── gui
│       │               ├── SampleIntegrationTest.java
│       │               └── SampleTest.java
│       └── resources
│           ├── log4j2.xml
│           └── tool.properties
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.5.3 Included frameworks/libraries

1. Like all of QBiC's JVM based projects, lib-java uses QBiC's parent-pom.
2. junit4 is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
3. Preconfigured ReadTheDocs.
4. Four Github workflows are shipped with the template
    1. `build_docs.yml`, which builds the ReadTheDocs documentation.
    2. `build_package.yml`, which builds the Maven based project.
    3. `java_checkstyle.yml`, which runs Checkstyle using Google's Styleguides.
    4. `run_test.yml`, which runs all junit4 tests.
    5. `qube_lint.yml`, which runs QUBE's linting on the project.
    6. `pr_to_master_from_development_only.yml` which fails if the PR does not come from a release or hotfix branch

14.5.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-java based projects.

14.6 portlet-groovy

14.6.1 Purpose

portlet-groovy is QBiC's template for portlets. Portlets are pluggable user interface software components, which are managed and displayed in a web portal.
They are a major part of QBIC's web presence.

14.6.2 Design

portlet-groovy follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
│   ├── requirements.txt
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── groovy_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
├── README.rst
├── .readthedocs.yml
├── src
```

(continues on next page)

(continued from previous page)

```
│   ├── main
│   │   ├── groovy
│   │   │   └── life.qbic.portal.portlet
│   │   │       └── SamplePortlet.groovy
│   │   ├── resources
│   │   │   ├── developer.properties.example
│   │   │   ├── .gitignore
│   │   │   ├── life
│   │   │   │   └── qbic
│   │   │   │       └── portal
│   │   │   │           └── portlet
│   │   │   │               └── AppWidgetSet.gwt.xml
│   │   │   ├── log4j2.xml
│   │   │   └── portlet.properties
│   │   └── webapp
│   │       ├── VAADIN
│   │       │   └── themes
│   │       │       └── mytheme
│   │       │           ├── addons.scss
│   │       │           ├── mytheme.scss
│   │       │           ├── styles.css
│   │       │           └── styles.scss
│   │       └── WEB-INF
│   │           ├── liferay-display.xml
│   │           ├── liferay-plugin-package.properties
│   │           ├── liferay-portlet.xml
│   │           ├── portlet.xml
│   │           └── web.xml
│   └── test
│       ├── java
│       │   └── life
│       │       └── qbic
│       │           └── portal
│       │               └── portlet
│       │                   ├── SamplePortletIntegrationTest.java
│       │                   └── SamplePortletTest.java
│       └── resources
│           ├── log4j2.xml
│           └── portlet.properties
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.6.3 Included frameworks/libraries

During the creation you will be asked whether or not you want to use the

1. openbis client. This will include the openbis-client-lib in your project.
2. openbis raw api. This will include the openbis-api in your project.
3. qbic databases. This will include the mariadb-java-client in your project.
4. vaadin charts. This will include the vaadin-charts in your project.

1. Like all of QBiC's JVM based projects, lib-java uses QBiC's parent-pom.

2. junit4 is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
3. Preconfigured ReadTheDocs.
4. Three Github workflows are shipped with the template
    1. `build_docs.yml`, which builds the ReadTheDocs documentation.
    2. `groovy_checkstyle.yml`, which runs npm-groovy-lint, which can be seen as a wrapper around CodeNarc.
    3. `run_test.yml`, which runs all junit4 tests.
    4. `qube_lint.yml`, which runs QUBE's linting on the project.
    5. `pr_to_master_from_development_only.yml` which fails iif the PR does not come from a release or hotfix branch

14.6.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-java based projects.

14.7 portlet-groovy-osgi

14.7.1 Purpose

portlet-groovy-osgi is QBiC's template for portlets. Portlets are pluggable user interface software components, which are managed and displayed in a web portal.
They are a major part of QBIC's web presence.
The portlet is OSGi ready. The framework, into which the portlet is plugged, is independent of the portlet that is added and makes it more robust to changes.

14.7.2 Design

portlet-groovy-osgi follows the standard Maven project layout.

```
├── AUTHORS.rst
├── CHANGELOG.rst
├── CODEOFCONDUCT.rst
├── .dependabot
│   └── config.yml
├── docs
│   ├── authors.rst
│   ├── changelog.rst
│   ├── codeofconduct.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   ├── Makefile
│   ├── readme.rst
│   ├── requirements.txt
```

(continues on next page)

(continued from previous page)

```
│   └── usage.rst
├── .github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug_report.md
│   │   ├── feature_request.md
│   │   └── general_question.md
│   ├── pull_request_template.md
│   └── workflows
│       ├── build_docs.yml
│       ├── build_package.yml
│       ├── groovy_checkstyle.yml
│       └── run_tests.yml
├── .gitignore
├── LICENSE
├── pom.xml
├── qube.cfg
├── .qube.yml
├── README.rst
├── .readthedocs.yml
├── src
│   ├── main
│   │   ├── groovy
│   │   │   └── life.qbic.portal.portlet
│   │   │       └── AppTheme.groovy
│   │   │       └── SamplePortletUI.groovy
│   │   ├── resources
│   │   │   ├── developer.properties.example
│   │   │   ├── .gitignore
│   │   │   ├── life
│   │   │   │   └── qbic
│   │   │   │       └── portal
│   │   │   │           └── portlet
│   │   │   │               └── AppWidgetSet.gwt.xml
│   │   │   ├── log4j2.xml
│   │   │   └── portlet.properties
│   │   └── webapp
│   │       ├── VAADIN
│   │       │   └── themes
│   │       │       └── mytheme
│   │       │           ├── addons.scss
│   │       │           ├── mytheme.scss
│   │       │           ├── styles.css
│   │       │           └── styles.scss
│   │       └── WEB-INF
│   │           ├── liferay-display.xml
│   │           ├── liferay-plugin-package.properties
│   │           ├── liferay-portlet.xml
│   │           ├── portlet.xml
│   │           └── web.xml
│   └── test
│       ├── java
│       │   └── life
│       │       └── qbic
│       │           └── portal
│       │               └── portlet
│       │                   ├── SamplePortletIntegrationTest.java
│       │                   └── SamplePortletTest.java
│       │
```

(continues on next page)

(continued from previous page)

```
│       └── resources
│           ├── log4j2.xml
│           └── portlet.properties
├── .travis.settings.xml
└── .travis.yml
```

If you are unfamiliar with specific files/file types, you may find them in our *Newbie Guide to QBiC software*.

14.7.3 Included frameworks/libraries

During the creation you will be asked whether or not you want to use the

1. openbis client. This will include the openbis-client-lib in your project.
2. openbis raw api. This will include the openbis-api in your project.
3. qbic databases. This will include the mariadb-java-client in your project.
4. vaadin charts. This will include the vaadin-charts in your project.

1. Like all of QBiC's JVM based projects, lib-java uses QBiC's parent-pom.
2. spock is currently QBiC's testing framework of choice. If you require mocking for any integration tests or advanced command line tests, Mockito may be useful.
3. Preconfigured ReadTheDocs.
4. Three Github workflows are shipped with the template
   1. `build_docs.yml`, which builds the ReadTheDocs documentation.
   2. `groovy_checkstyle.yml`, which runs npm-groovy-lint, which can be seen as a wrapper around CodeNarc.
   3. `run_test.yml`, which runs all junit4 tests.
   4. `qube_lint.yml`, which runs QUBE's linting on the project.
   5. `pr_to_master_from_development_only.yml` which fails if the PR does not come from a release or hotfix branch

14.7.4 Usage

The main Maven commands such as `mvn test`, `mvn verify`, `mvn package` and more are used to test and package cli-java based projects.

14.8 Shared FAQ

14.8.1 How do I setup Read the Docs?

qube ships with a full, production ready Read the Docs setup. You need to import your documentation on Read the Docs website. Do not forget to sync your account first to see your repository.

14.8.2 What is Dependabot and how do I set it up?

Dependabot is a service, which (for supported languages) automatically submits pull requests for dependency updates. qube templates ship with dependabot configurations, if the language is supported by Dependabot. To enable Dependabot you need to login (with your Github account) and add your repository (or enable Dependabot for all repositories). Note that you need to do this for every organization separately. Dependabot will then pick up the configuration and start submitting pull requests!

14.8.3 How do I add a new template?

Please follow *Adding new templates*.

CHAPTER FIFTEEN

ADDING NEW TEMPLATES

Adding new templates is one of the major improvements to qube, which is why we are dedicating a whole section to it. Please note that creating new templates is a time consuming task. So be prepared to invest a few hours to bring a new template to life. The integration into qube however, is straight forward if you follow the guide below. Due to the tight coupling of our templates with all qube commands such as `create`, `list`, `info`, `lint` and `bump-version`, new templates require the modification of several files.

qube uses cookiecutter to create all templates. You need to familiarize yourself beforehand with cookiecutter to able to write templates, but don't worry, it's pretty easy and you usually get by with very few cookiecutter variables. You can start with you very first cookiecutter template and then simply see how the other existing qube templates are made and copy what you need.

The following sections will line out the requirements for new templates and guide you through the process of adding new templates step by step. Nevertheless, we strongly encourage you to discuss your proposed template first with us in public *via* a Github issue.

15.1 Template requirements

To keep the standard of our templates high we enforce several standards, to which all templates **must** adhere. Exceptions, where applicable, but they would have to be discussed beforehand. Hence, the term *should*.

1. New templates should be novel. We do not want a second cli-java template, but you are of course always invited to improve it. A new commandline library does not warrant an additional template, but rather modifications of the existing template with cookiecutter if statements. However, distinct modifications of already existing templates may be eligible. An example would be to add a GUI template for a language, which does not yet have a GUI template. Templates for domains, which we do not yet cover or additional languages to already existing domains are of course more than welcome.
2. All templates must adhere to QBiC's version standards. For example all JVM based templates must use the QBiC's Java version (which is currently 8).
3. All templates should build as automatically as possible and download all dependencies without manual intervention.
4. All templates should have a testing and possibly mocking framework included.
5. All templates should provide a readthedocs setup (include changelog and a codeofconduct), a README.rst file, a LICENSE, Github issue and pull request templates and a .gitignore file. Moreover, a .dependabot configuration should be present if applicable. Note that most of these are already included in our common_files and do not need to be rewritten. More on that below.
6. All templates must implement all required functionality to allow the application of all commands mentioned above to them, which includes a qube.cfg file, the template being in the available_templates.yml and more.
7. All templates should have Github workflows, which at least build the documentation and the project.

8. Every template should also have a workflow inside qube, which creates a project from the template with dummy values.

15.2 Step by step guide to adding new templates

Let's assume that we are planning to add a new commandline Brainfuck template to qube. We discussed our design at length with the core team and they approved our plan. For the sake of this tutorial **we assume that the path / always points to /qube**. Hence, at this level we see `cli.py` and a folder per CLI command.

1. Let's add our brainfuck template information to `/create/templates/available_templates.yml` below the `cli` section.

```
cli:
    brainfuck:
        name: Brainfuck Commandline Tool
        handle: cli-brainfuck
        version: 0.0.1
        available libraries: none
        short description: Brainfuck Commandline tool with ANSI coloring
        long description: Amazing brainfuck tool, which can even show pretty unicorns␣
↪in the console.
            Due to ANSI coloring support they can even be pink! Please someone send␣
↪help.
```

2. Next, we add our brainfuck template to `/create/templates`

   Note that it should adhere to the standards mentioned above and include all required files. Don't forget to add a qube.cfg file to facilitate bump-version. See *Configuration* for details. It is **mandatory** to name the top level folder `{{ cookiecutter.project_slug }}`, which ensures that the project after creation will have a proper name. Furthermore, the `cookiecutter.json` file should have at least the following variables:

```
{
"full_name": "Homer Simpson",
"email": "homer.simpson@posteo.net",
"project_name": "sample-cli",
"project_slug": "sample-cli",
"version": "1.0.0",
"project_short_description": "Command-line utility to...",
"github_username": "homer_github"
}
```

The file tree of the template should resemble

```
├── cookiecutter.json
└── {{ cookiecutter.project_slug }}
    ├── docs
    │   ├── installation.rst
    │   └── usage.rst
    ├── .github
    │   └── workflows
    │       └── build_brainfuck.yml
    ├── hello.bf
    ├── qube.cfg
    └── README.rst
```

3. Now it is time to subclass the `TemplateCreator` to implement all required functions to create our template!

Let's edit `/create/domains/cli_creator.py`. Note that for new domains you would simply create a new file called DomainCreator.

In this case we suggest to simply copy the code of an existing Creator and adapt it to the new domain. Your new domain may make use of other creation functions instead of `create_template_without_subdomain`, if they for example contain subdomains. You can examine `create/TemplatorCreator.py` to see what's available. You may also remove functions such as the creation of common files.

If we have any brainfuck specific cookiecutter variables that we need to populate, we may add them to the TemplateStructCli.

Our brainfuck templates does not have them, so we just leave it as is.

For the next step we simply go through the `CliCreator` class and add our brainfuck template where required. Moreover, we implement a `cli_brainfuck_options` function, which we use to prompt for template specific cookiecutter variables.

```
@dataclass
class TemplateStructCli(MlfcoreTemplateStruct):
    """
    Intended Use: This class holds all attributes specific for CLI projects
    """

    """____BRAINFUCK___"""


class CliCreator(TemplateCreator):

    def __init__(self):
        self.cli_struct = TemplateStructCli(domain='cli')
        super().__init__(self.cli_struct)
        self.WD = os.path.dirname(__file__)
        self.WD_Path = Path(self.WD)
        self.TEMPLATES_CLI_PATH = f'{self.WD_Path.parent}/templates/cli'

        ''"" TEMPLATE VERSIONS ""'
        self.CLI_BRAINFUCK_TEMPLATE_VERSION = super().load_version('cli-brainfuck')

    def create_template(self, dot_qube: dict or None):
        """
        Handles the CLI domain. Prompts the user for the language, general and domain␣
↪specific options.
        """

        self.cli_struct.language = qube_questionary_or_dot_qube(function='select',
                                                                 question='Choose the␣
↪project\'s primary language',
                                                                 choices=['brainfuck'],
                                                                 default='python',
                                                                 dot_qube=dot_qube,
                                                                 to_get_property=
↪'language')

        # prompt the user to fetch general template configurations
        super().prompt_general_template_configuration(dot_qube)

        # switch case statement to prompt the user to fetch template specific␣
↪configurations
        switcher = {
            'brainfuck': self.cli_brainfuck_options
```

(continues on next page)

(continued from previous page)

```
        }
        switcher.get(self.cli_struct.language)(dot_qube)

        self.cli_struct.is_github_repo, \
            self.cli_struct.is_repo_private, \
            self.cli_struct.is_github_orga, \
            self.cli_struct.github_orga \
            = prompt_github_repo(dot_qube)

        if self.cli_struct.is_github_orga:
            self.cli_struct.github_username = self.cli_struct.github_orga

        # create the chosen and configured template
        super().create_template_without_subdomain(f'{self.TEMPLATES_CLI_PATH}')

        # switch case statement to fetch the template version
        switcher_version = {
            'brainfuck': self.CLI_BRAINFUCK_TEMPLATE_VERSION
        }
        self.cli_struct.template_version, self.cli_struct.template_handle = switcher_
↪version.get(
            self.cli_struct.language.lower()), f'cli-{self.cli_struct.language.
↪lower()}'

        super().process_common_operations(domain='cli', language=self.cli_struct.
↪language, dot_qube=dot_qube)

        [...]

    def cli_brainfuck_options(self):
        """ Prompts for cli-brainfuck specific options and saves them into the␣
↪MlfcoreTemplateStruct """
        pass
```

4. If a new template were added we would also have to import our new Creator in `create/create.py` and add the new domain to the domain prompt and the switcher.
   However, in this case we can simply skip this step, since `cli` is already included.

```
def choose_domain(domain: str):
    """
    Prompts the user for the template domain.
    Creates the .qube.yml file.
    Prompts the user whether or not to create a Github repository
    :param domain: Template domain
    """
    if not domain:
        domain = click.prompt('Choose between the following domains',
                              type=click.Choice(['cli']))

    switcher = {
        'cli': CliCreator,
    }

    creator_obj = switcher.get(domain.lower())()
    creator_obj.create_template()
```

5. Linting is up next! We need to ensure that our brainfuck template always adheres to the highest standards!

Let's edit `lint/domains/cli.py`.

We need to add a new class, which inherits from TemplateLinter and add our linting functions to it.

```
class CliBrainfuckLint(TemplateLinter, metaclass=GetLintingFunctionsMeta):
    def __init__(self, path):
        super().__init__(path)

    def lint(self):
        super().lint_project(self, self.methods)

    def brainfuck_files_exist(self) -> None:
        """
        Checks a given pipeline directory for required files.
        Iterates through the templates's directory content and checkmarks files for␣
↪presence.
        Files that **must** be present::
            'hello.bf',
        Files that *should* be present::
            '.github/workflows/build_brainfuck.yml',
        Files that *must not* be present::
            none
        Files that *should not* be present::
            none
        """

        # NB: Should all be files, not directories
        # List of lists. Passes if any of the files in the sublist are found.
        files_fail = [
            ['hello.bf'],
        ]
        files_warn = [
            [os.path.join('.github', 'workflows', 'build_brainfuck.yml')],
        ]

        # List of strings. Fails / warns if any of the strings exist.
        files_fail_ifexists = [

        ]
        files_warn_ifexists = [

        ]

        files_exist_linting(self, files_fail, files_fail_ifexists, files_warn, files_
↪warn_ifexists)
```

We need to ensure that our new linting function is found when linting is applied. Therefore, we turn our eyes to `lint/lint.py`, import our CliBrainfuckLinter and add it to the switcher.

```
from qube.lint.domains.cli import CliBrainfuckLint

switcher = {
    'cli-brainfuck': CliBrainfuckLint,
}
```

Our shiny new CliBrainfuckLinter is now ready for action!

6. The only thing left to do now is to add a new Github Actions workflow for our template. Let's go one level up in the folder tree and create `.github/workflows/create_cli_brainfuck.yml`.

We want to ensure that if we change something in our template, that it still builds!

```
name: Create cli-brainfuck Template

on: [push]

jobs:
  build:

      runs-on: ubuntu-latest
      strategy:
        matrix:
          python: [3.7, 3.8]

      steps:
      - uses: actions/checkout@v2
        name: Check out source-code repository

      - name: Setup Python
        uses: actions/setup-python@v1
        with:
          python-version: ${{ matrix.python }}

      - name: Build qube
        run: |
          python setup.py clean --all install

      - name: Create cli-brainfuck Template
        run: |
          echo -e "\n\n\n\n\nn\n\n\n\nn" | qube create

      - name: Build Package
        uses: fabasoad/setup-brainfuck-action@master
        with:
          version: 0.1.dev1
      - name: Hello World
        run: |
          brainfucky --file ExplodingSpringfield/hello.bf


We were pleasently surprised to see that someone already made a Github Action for␣
↪brainfuck.
```

8. Finally, we add some documentation to `/docs/available_templates.rst` and explain the purpose, design and frameworks/libraries.

   That's it! We should now be able to try out your new template using `qube create` The template should be creatable, it should automatically lint after the creation and Github support should be enabled as well! If we run `qube list` Our new template should show up as well! I'm sure that you noticed that there's not actually a brainfuck template in qube (yet!).

   To quote our mighty Math professors: 'We'll leave this as an exercise to the reader.'

CHAPTER

SIXTEEN

TROUBLESHOOTING

All currently known issues can be found on our Github issue tracker. If there are any major known issues they will be listed here.

CHAPTER

SEVENTEEN

CHANGELOG

This project adheres to Semantic Versioning.

17.1 2.6.1 (2020-11-06)

Added

- Add report generation script to common files

Fixed

Dependencies

Deprecated

17.2 2.6.0 (2020-10-27)

Added

- Add template for OSGi Groovy library bundles
- Add template for OSGi Groovy portlet bundles

Fixed

- Fix missing license property bug, that showed up if the license placeholder was referenced in a template.

Dependencies

Deprecated

- Java 8, templates now build with JDK 11.

17.3 2.5.1 (2020-10-16)

Added

Fixed

- qube lint now wraps too long lines

Dependencies

Deprecated

17.4 2.5.0 (2020-10-06)

Added

- verbose support #186

Fixed

- sync workflow now polls instead of being triggered on push #170
- renamed branch protection workflow #190
- refactored sync command
- Faster build time by fixing the order of Maven repositories for dependency resolving
- Ignore rule for Vaadin widgetsets

Dependencies

Deprecated

17.5 2.4.6 (2020-10-02)

Added

Fixed

- Fix missing properties for portlet domain
- Fix #169

Dependencies

Deprecated

17.6 2.4.5 (2020-10-02)

Added

- Ignores additional Maven files

Fixed

- Preserve boolean case when loading YAML boolean values
- Force push changes to the TEMPLATE branch during sync

Dependencies

Deprecated

17.7 2.4.4 (2020-10-02)

Added

Fixed

- Fix the pull request creation after updating syncing the TEMPLATE branch. Qube reported a `FileNotFoundError` for the sync workflow file, because it tried to access this file in an empty directory.
- Removed redundant sync_workflow workarounds
- sync and maven test workflow yaml syntax

Dependencies

Deprecated

17.8 2.4.3 (2020-10-01)

Added

Fixed

- Sets correct repo owner for the `qube sync`

Dependencies

Deprecated

17.9 2.4.2 (2020-10-01)

Added

- Enables debug logging

Fixed

Dependencies

Deprecated

17.10 2.4.1 (2020-10-01)

Added

Fixed

Dependencies

- Updated parent pom to 3.1.1
- Updated template versions to 1.0.1

Deprecated

17.11 2.4.0 (2020-10-01)

Added

- Now using Johnny5 for the sync workflow by default
- Maven caching for tests

Fixed

- Add all *src/main/webapp/VAADIN/widgetsets* folders to *.gitignore*
- Makefile now uses pip instead of setup.py by default

Dependencies

Deprecated

17.12 2.3.0 (2020-09-28)

Added

- Added release deployment GA workflow for JVM templates
- Added workflow to build software reports and internal documentation

Fixed

- Fixed parent-pom version being outdated -> 3.1.0
- Fixed further outdated dependencies in various poms
- Fixed release URL in all poms
- Allow PR from 'hotfix' branches

Dependencies

Deprecated

- Removed PR allowance from patch branches
- Removed Travis CI support

17.13 2.2.0 (2020-08-21)

Added

Fixed

- Couple of docs fixes
- Now always using hyphens for options

Dependencies

Deprecated

17.14 2.1.0 (2020-08-21)

Added

- Option to config –view to get the current set configuration
- Option –set_token to set the sync token again
- Sync docs improved
- Support for QUBE TODO: and TODO QUBE:

Fixed

- Sync for organization repositories

Dependencies

Deprecated

17.15 2.0.0 (2020-08-17)

Added

- Strong code refactoring overhauling everything
- Added config command to recreate config files
- Added upgrade command to update qube itself
- Added sync command to sync a qube project
- Help messages are now custom
- Bump-version lints versions before updating
- Added a metaclass to fetch all linting functions
- Master requires PR review & no stale PRs
- Greatly improved the documentation
- Much more…

Fixed

- PR check WF now correctly requires PRs to master to be from *patch* or *release* branches

Dependencies

- Too many updates to jot down…!

Deprecated

17.16 1.4.1 (2020-05-23)

Added

Fixed

- Reverted simplified common files copying, since it broke Github support

Dependencies

Deprecated

17.17 1.4.0 (2020-05-23)

Added

- Added Rich for tracebacks & nice tables
- New ASCII Art!

Fixed

Dependencies

Deprecated

17.18 1.3.2 (2020-05-22)

Added

- Strongly simplified common files copying
- info now automatically reruns the most similar handle

Fixed

Dependencies

Deprecated

17.19 1.3.1 (2020-05-20)

Added

- Checking whether project already exists on readthedocs

Fixed

- bump-version SNAPSHOT handling strongly improved

Dependencies

- requests==2.23.0 added
- packaging==20.4 added

Deprecated

17.20 1.3.0 (2020-05-20)

Added

- bump-version now supports SNAPSHOTS
- documentation about 4 portlet prompts
- new COOKIETEMPLE docs css

Fixed

- Tests GHW names

Dependencies

Deprecated

17.21 1.2.1 (2020-05-03)

Added

- Refactored docs into common files

Fixed

Dependencies

Deprecated

17.22 1.2.0 (2020-05-03)

Added

- QUBE linting workflow for all templates
- PR to master from development only WF
- custom COOKIETEMPLE css

Fixed

- setup.py development status
- max width for docs for all templates
- PyPi badge is now green

Dependencies

- flake 3.7.9 -> 3.8.1

Deprecated

17.23 1.1.0 (2020-05-03)

Added

- The correct version tag :)

Fixed

- Readthedocs width is now

Dependencies

Deprecated

17.24 1.0.0 (2020-05-03)

Added

- Created the project using COOKIETEMPLE
- Added create, list, info, bump-version, lint based on COOKIETEPLE
- Added cli-java template
- Added lib-java template
- Added gui-java template
- Added service-java template
- Added portlet-groovy template

Fixed

Dependencies

Deprecated

CHAPTER

EIGHTEEN

CREDITS

18.1 Development Lead

- Lukas Heumos <lukas.heumos@posteo.net>

18.2 Contributors

All of QBiC Software

CHAPTER

NINETEEN

CONTRIBUTOR COVENANT CODE OF CONDUCT

19.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

19.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

19.3 Our Responsibilities

Maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

19.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

19.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

19.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search