QTO DEVOPS GUIDE

1. GUIDING PRINCIPLE'S

This section might seem too philosophical for a start, yet all the development in the qto has ATTEMPTED to follow the principles described bellow. If you skip this section now you might later on wander many times why something works and it is implemented as it is ... and not "the right way".
Of course you are free to not follow these principles, the less you follow them the smaller the possibility to pull features from your instance(s) - you could even use the existing functionality to create a totally different fork with different name and start developing your own toll with name X - the authors give you the means to do that with this tool ... , but if you want to use and contribute to THIS tool than you better help defined those leading principles and follow them.

1.1. Use common sense

Use common sense when applying all those principles. Of course they are not engraved in stone and you should be flexible enough for the actual situation, problem, issue etc.

1.2. Personal responsibility

Any given instance of the qto should have ONE and only ONE person which is responsible at the end for the functioning of THE instance - so think carefully before attempting to take ownership of an instance. The author(s) of the code are not responsible for the operation, bugs or whatever happens to a new instance. As a responsible owner of an instance you could create, share and assign issues to the authors of the source code, yet there is no Service Level Agreement, not even promise to help.

1.3. Attempt for 100% test coverage to achieve reliability

The more you increase your test coverage the greater the confidence that the code will work as expected.
Do not write a single function without first implementing the testing call for that function - this has been proven really, really difficult, yet the more features are added the less the time wasted in troubleshooting of bugs and un-expected behaviour when proper testing is implemented.
Testing ensures the consistency and future expandability of the functionalities. Our velocity increases with the WORKING features and functionalities added over time. ANYTHING, which is not working, or not even sure about how it should be working MUST be fixed/deleted.

1.4. It should just work

Any instance of the qto should simply work as promised - no less, no more. Any instance is the combination of code, configurations, binaries in the System and data - that is the instance you are using should just work for the set of functionalities promised.

1.5. Naming conventions

All the names used in the code and the configurations MUST BE human readable and expandable - that is name the objects from the greater realm to the smaller - for example <<env>>_<<db_name>> , because the concept of operational IT environments (dev , test , qas , prd) is broader than the concept of a application databases ...

1.6. Be friendly to all

Especially to technical personnel, as you cannot achieve user-friendliness for the end-users unless your developers and technical personnel are happy while interacting with your artefacts.

1.7. Aim for simplicity

Things should be as simple as possible, but not simpler - if Einstein said it it makes sense - having lost so much time in endless loops of IT complexity - the older we get the more it gets more rational.

1.8. Do not allow broken windows

A broken windows is any peace of code or documentation which is hanging around not included in the integration tests suite and not matching the most up-to-date standards for work deliverables. Either bring it up to the standard level or get rid of it.
As soon as you find a bug, write a test for it, if you can't create the needed testing setup invest in time developing the needed skills.

1.9. Do not add a commit without passing ALL the regression tests

Even in your personal branch. Really. Because after the application has surpassed the mark of having 200 000 lines of code the complexity added to a "broken machine" WILL NOT justify the breaking of an existing feature. If you do not consider the feature / functionality tested as important than feel free to REMOVE it (both implementations AND tests) in that very same commit.

2. SYSADMIN ETL OPERATIONS

2.1. Backup a database table

You backup a database table with the following one-liner. Noe

```
# obs you have to have the shell vars pre-loaded !!!
bash src/bash/qto/qto.sh -a backup-postgres-table -t
my_table
```

2.2. Run increase-date action

You track the issues of your projects by storing them into xls files in "daily" proj_txt dirs.
Each time the day changes by running the increase-date action you will be able to clone the data of the previous date and start working on the current date.

```
bash src/bash/qto/qto.sh -a increase-date
```

2.3. Load xls issues to db and from db to txt files

to load xls issues to db and from db to txt files

```
bash src/bash/qto/qto.sh -a xls-to-db -a db-to-txt

# or run for all the periods
for period in `echo daily weekly monthly yearly`; do export
period=$period ;
bash src/bash/qto/qto.sh -a xls-to-db -a db-to-txt ; done ;
```

2.4. Run the qto file to db load

Run the qto file to db load

```
# ensure the following actions will be tested
cat src/bash/qto/tests/run-qto-tests.lst | grep -v '#'
# output should be if not correct
check-perl-syntax
run-qto

# test those uncommented actions
bash src/bash/qto/test-qto.sh
```

2.5. Verify the inserted data from the db

Verify the inserted data from the db as follows:

```
# check that the rows where inserted
echo 'SELECT * FROM issue ; ' | psql -d dev_qto
```

2.6. Dirs naming conventions

The dir structure should be logical and a person navigating to a dir should almost understand what is to be find in thre by its name ..

3. NAMING CONVENTIONS

3.1. Product instance directories

3.2. Root Dirs naming conventions

The root dirs and named as follows:
bin - contains the produced binaries for the project
cnf - for the configuration
dat - for the data of the app
lib - for any external libraries used
src - for the source code of the actual projects and subprojects

3.3. Dabase naming conventions

Each database must start with it's environment prefix - dev, tst or prd. And yes this is so fundamentally in-build into qto that changing this naming convention will definitely destroy your application.

4. SOURCE CODE MANAGEMENT

The qto is a derivative of the wrapp tool - this means that development and deployment process must be integrated into a single pipeline.

4.1. Configure and use git ALWAYS by using ssh identities

You probably have access to different corporate and public git repositories. Use your personal ssh identity file you use in GitHub to push to the qto project. The following code snippet demonstrates how you could preserve your existing git configurations (even on corporate / intra boxes) , but use ALWAYS the personal identity to push to the qto...

```
# create the company identity file
ssh-keygen -t rsa -b 4096 -C "first.last@corp.com"

# save private key to ~/.ssh/id_rsa.corp,
cat ~/.ssh/id_rsa.corp.pub

# copy paste this string into your corp web ui security ssh
keys

# create your private identify file
ssh-keygen -t rsa -b 4096 -C "me@gmail.com"
# save private key to ~/.ssh/id_rsa.me, note the public key
~/.ssh/id_rsa.me.pub
cat ~/.ssh/id_rsa.me.pub # copy paste this one into your
githubs, private keys

# set alias for the git command to avoid overtyping ...
alias git='GIT_SSH_COMMAND="ssh -i ~/.ssh/id_rsa.ysg " git'

# clone a repo
git clone git@git.in.corp.com:corp/project.git

export git_msg="my commit msg with my corporate identity,
explicitly provide author"
git add --all ; git commit -m "$git_msg" --author "MeFirst
MeLast <first.last@corp.com>"
git push
# and verify
clear ; git log --pretty --format='%h %ae %<(15)%an ::: %s
```

4.2. Aim for traceability between user-stories, requirements, features and functionalities

Once the issues are defined and you start working on your own branch which is named by the issue-id aim to map one on one each test in your code with each listed requirement in confluence and / or JIRA.

4.3. Zero tolerance for bugs, especially crashes

As soon as bugs are identified and reproduceable, register them as issues and resolve them with prio (aka priority) 1.
After resolution, think about the root cause of the bug, the mear fact that the bug occurred tells that something in the way of working has to be improved , what ?!
Bugs are like broken windows the more you have them the faster the value of your building will be down to zero.

4.4. ALWAYS Start with a test unit creation

Do not ever never write code without starting first the unit test on how-to test the code. Period.
This is he only way to avoid braking old functionalities when the application code base grows larger.
Each time a new bug is found fix it by adding new Unit Test!

4.5. Branch for COMMON development - dev

No code should be merged into the development branch without broad testing coverage and approval from the owner of the instance - as the owner of the instance is at the end responsible personally for the whole instance, since once a change has been merged to develop it must pass as quickly as possible to tst, prd and master.

4.6. Integration testing in the tst branch

The tst branch is dedicated for integration testing of all the tests, the deployment, performance testing and configuration changes. Should you need to perform bigger than a small hotfix changes you must branch the tst branch into a separate dev--feature branch and re-run the integration testing and approval all over.
At the end all the integration tests should be behind this shell call.

```
bash src/bash/qto/qto.sh -a run-integration-tests
```

4.7. Production in the prd branch

The prd branch is the one deployed to the production environment. This code is NOT straight merged into the master branch , but after certain time depending on the dynamic of the tool with bugless operation merged.

4.8. master branch - the single truth for current stable version of the software

Once the business has approved a new version - it should be moved to the master branch and all other branches including the separate feature branches MUST be REBASED (and NOT MERGED !!!) from the master branch to accommodate any hotfixes, configuration related adjustments or quick bug fixes detected in production only.

5. WAY OF WORKING

This section describes the way of working within a team working on the qto project.
The work on the qto project is conducted by using the Scrum methodology, thus the Scrum

5.1. Definition of Done

Each issue must have a tangible artifact. An issue without tangible artifact is a thought thrown in the air.
The DoD must be iterated and updated during each Sprint Review.

5.2. Chat / IRC

Should you want a quicker respond than 2 hours use a chat tool. Do not expect people to answer you straight away, it takes 5 to 20 min to reach the most productive flow state, thus not answering your question might be the more productive option from the point of view of the organisation.

5.3. E-mail communication

Do not use e-mail communication for code style, testing, developing etc. Issues which could be achieved with the code review interface of the source code management system.
Before writing an e-mail think first could you find a way to avoid writing it at all.
Do not expect answer of your e-mail within 2 hours.
Use e-mail when you have to get an written evidence on agreed matters, which might cause later on discussions.

5.4. Documentation

Undocumented feature is not a feature.

5.5. Issues management

At the end of the month you should move the completed issues to the yearly_issues table as follows:

```
psql -d tst_qto -c "
INSERT INTO yearly_issues
( guid  , id  , type  , category  , status  , prio  , name
, description  , owner  , update_time)
SELECT
guid  , id  , type  , category  , status  , prio  , name  ,
description  , owner  , update_time
FROM monthly_issues WHERE 1=1 and status='09-done'
ON CONFLICT (id) DO UPDATE
SET
guid = excluded.guid ,id = excluded.id ,type = excluded.type
,category = excluded.category ,status = excluded.status
,prio = excluded.prio, name = excluded.name ,description =
excluded.description ,owner = excluded.owner ,update_time =
excluded.update_time
;
"
```

6. FEATURE IMPLEMENTATION WORKFLOW

This section describes the common workflow for implementing a feature.
As in other places the main principle to follow is "use common sense" , thus try to follow this workflow for feature implementation, but challenge it as soon as it defies the common sense.

6.1. Issue creation

Even if you do not have a defined documentation artifact - create a new issue, which could be the start for a an action affecting the run-state, configuration , data , features and functionalities or other aspects of the qto application.
An issue could be a bug, a request for a feature or even simply an undefined combination of problems and solution which could quickly be formalized by defining a new requirement, another issue, feature-request

6.2. User-Story creation

Use the following template while creating the user story:
As an <<role>>
In order to <<achieve something>>
I wanto to be able <<action-description>>

6.3. Requirements creation

Depending on the size and agility of your organisation formal requirements exist.

6.4. Problem registration

Problems are usually entities which last for longer time period.

6.5. Feature branch creation

Create the feature branch by using the following naming convention:
 - dev--<<short-feature-title>>

Even the smallest proof of concept needs a small test-entry point. Start always with the testing and the testing scalability in mind.

```
git branch -a
* dev
dev--qto-18050801-add-order-by-in-select-ctrlr
master
prd
tst
remotes/origin/dev
remotes/origin/master
```

6.6. Create a test-entry point

6.7. Implementation of Proof of Concept

Aim to create a small POC for the new concept, feature or functionality - for example a page having a lot of hardcoding, which constrains the scope for ONLY this new thing.
Strive however to use the same naming convention, and implement with future integrations within the end truly dynamic code.

6.8. Prototype implementation

The same instructions as the POC apply, but the prototype contains a certain and broader level of integration with the dynamic parts of the System.

6.9. Unit and / or integration test creation

Strive to create always unit and / or integration test(s).

6.10. Implementation

Implement by quick unit test runs. Constantly improve both the code , configuration changes and the test code.

6.11. Deployment and test to the test environment

Deploy to the test environment.

```
# deploy to the tst environment
bash src/bash/qto/qto.sh -a to-tst

# go to the product instance dir of the tst env for this
version
cd ../qto.<<version>>.tst.<<owner>>
bash src/bash/qto/qto.sh -a run-integration-tests
bash src/bash/qto/qto.sh -a run-functional-tests
```

6.12. Deployment and test to the production environment

Repeat the same to the production environment. As the current version is usually work in progress your stable version will be one level bellow and thanks to the architecture of the tool you could test in the production environment (as soon as you have proper configuration).

6.13. Quality assurance iteration

This phase might be longer depending on the feature. Some of the features stay in quality assurance mode EVEN if they have been deployed to production.

6.14. DoD check-list walkthrough

Perform the DoD checklist as follows.

6.15. The feature or functionality current description is added in the docs

The feature or functionality current description is added in the Features and Functionalities document.

6.16. The related requirement is added in the requirements document

The related requirement is added in the requirements document - there might be one or more requirements added.

6.17. At least 2 times passed functional and js tests run

Use the following shell actions (Note that since v0.6.7 as authentication is required for most of the web-actions the QTO_ONGOING_TEST environmental variable has to be set to 1 to run those separately) :

```
bash src/bash/qto/qto.sh -a run-js-tests
bash src/bash/qto/qto.sh -a run-functional-tests
```

6.18. At least 2 times passed integration tests in each environment instance

At least 2 times passed unit tests run in each environment instance - run the unit tests at least twice per environment. Should the run behave differently start all over from dev. Since v0.6.7 as authentication is required for most of the web-actions the QTO_ONGOING_TEST environmental variable has to be set to 1 to run those separately.

```
bash src/bash/qto/qto.sh -a run-integration-tests
```

6.19. Deployment to the test environment

Deploy to the test environment as shown in the code snippet bellow. Re-run the tests via the tests shell actions.

```
# deploy to the tst environment
bash src/bash/qto/qto.sh -a to-tst

# go to the product instance dir of the tst env for this
version
cd ../qto.<<version>>.tst.<<owner>>
```

6.20. Check that all the files in the deployment package are the same as those in the latest commit of the dev git branch.

Deploy to the test environment as follows:

Well just chain the both commands.

```
# deploy to the tst environment
bash src/bash/qto/qto.sh -a to-tst

# go to the product instance dir of the tst env for this
version
cd ../qto.<<version>>.tst.<<owner>>
```

6.21. restart the application layer

```
bash src/bash/qto/qto.sh -a mojo-morbo-stop ; bash
src/bash/qto/qto.sh -a mojo-morbo-start
```

7. KNOWS ISSUES AND WORKAROUNDS

7.1. All tests fail with the 302 error

This one is actually a bug ... all the tests not requiring non-authentication mode should set it in advance ...

```
# disable authentication during testing
export QTO_ONGOING_TEST=1

# call the test once again
perl src/perl/qto/t/lib/Qto/Controller/TestHiCreate.t
```

7.2. Morbo is stuck

7.2.1. Problem description

This one occurs quite often , especially when the application layer is restarted, but the server not

```
# the error msg is
 [INFO ] 2018.09.14-10:23:14 EEST [qto][@host-name] [4426]
running action :: mojo-morbo-start:doMojoMorboStart
(Not all processes could be identified, non-owned process
info
 will not be shown, you would have to be root to see it
all.)
tcp        0      0 0.0.0.0:3001            0.0.0.0:*
LISTEN      6034/qto
tcp        0      0 0.0.0.0:3002            0.0.0.0:*
LISTEN      7626/qto
Can't create listen socket: Address already in use at
/usr/local/share/perl/5.26.0/Mojo/IOLoop.pm line 130.
 [INFO ] 2018.09.14-10:23:16 EEST [qto][@host-name] [4426]
STOP FOR qto RUN with:
 [INFO ] 2018.09.14-10:23:16 EEST [qto][@host-name] [4426]
STOP FOR qto RUN: 0 0 # = STOP MAIN = qto
qto-dev ysg@host-name [Fri Sep 14 10:23:16]
[/vagrant/opt/csitea/qto/qto.0.4.9.dev.ysg] $
```

7.2.2. Probable root cause

This one occurs quite often , especially when the application layer is restarted, but the server not

```
# the error msg is
 [INFO ] 2018.09.14-10:23:14 EEST [qto][@host-name] [4426]
running action :: mojo-morbo-start:doMojoMorboStart
(Not all processes could be identified, non-owned process
info
 will not be shown, you would have to be root to see it
all.)
tcp        0      0 0.0.0.0:3001            0.0.0.0:*
LISTEN      6034/qto
tcp        0      0 0.0.0.0:3002            0.0.0.0:*
LISTEN      7626/qto
Can't create listen socket: Address already in use at
/usr/local/share/perl/5.26.0/Mojo/IOLoop.pm line 130.
 [INFO ] 2018.09.14-10:23:16 EEST [qto][@host-name] [4426]
STOP FOR qto RUN with:
 [INFO ] 2018.09.14-10:23:16 EEST [qto][@host-name] [4426]
STOP FOR qto RUN: 0 0 # = STOP MAIN = qto
qto-dev ysg@host-name [Fri Sep 14 10:23:16]
[/vagrant/opt/csitea/qto/qto.0.4.9.dev.ysg] $
```

7.2.3. Kill processes

List the running perl processes which run the morbo and kill the instances

```
ps -ef | grep -i perl

# be carefull what to kill
kill -9 <<proc-I-know-is-the-one-to-kill>>
```