MAINTENANCE GUIDE

1. GUIDING PRINCIPLE'S
1.1. Attempt for 100% test coverage to achieve reliability
1.2. Personal responsibility
2. COMMON OPERATIONS
2.1. To check the status of the postgreSql
2.2. To stop the postgreSql
2.3. To start the postgreSql
2.4. to check the port on which it is listening
2.5. Check the postgres status
2.6. Application Layer run-state management
2.6.1. start the application layer
2.6.2. stop the application layer
2.6.3. Restart OS network service
2.7. Security related operations
2.7.1. Add, modify and delete new users to the application
2.7.2. Regular users visibility
3. BACKUP AND RESTORE
3.1. Load database connectivity configuration securely
3.2. Backup a database
3.3. Backup a database table
3.4. Restore a database
3.5. Restore a database table
4. USAGE SCENARIOS
4.1. Run increase-date action
4.2. Verify the inserted data from the db
4.3. Load xls issues to db and from db to txt files
5. NAMING CONVENTIONS
5.1. Dirs naming conventions
5.2. Root Dirs naming conventions
6. SOURCE CODE MANAGEMENT
6.1. Configure and use git ALWAYS by using ssh identities
6.2. Aim for traceability between user-stories, requirements, features and functionalities
6.3. Zero tolerance for bugs
6.4. ALWAYS Start with a test unit creation
6.5. Branch for development - dev
6.6. Testing and integrations in the tst branch
6.7. Quality assurance in the qas branch
6.8. Production in the prd branch
7. WAY OF WORKING
7.1. Definition of Done
7.2. E-mail communication
7.3. Chat / IRC
7.4. Documentation
7.5. Issues management
8. FEATURE IMPLEMENTATION WORKFLOW
8.1. Issue creation
8.2. User-Story creation
8.3. Requirements creation
8.4. Problem registration
8.5. feature branch creation
8.6. Create a test-entry point
8.7. Implementation of Proof of Concept
8.8. Prototype implementation
8.9. Unit and / or integration test creation
8.10. Implementation
8.11. Deployment and test to the test environment
8.12. Deployment and test to the production environment
8.13. Quality assurance iteration
8.14. DoD check-list walkthrough
8.15. The feature or functionality current description is added in the docs
8.16. The related requirement is added in the requirements document
8.17. At least 2 times passed functional and js tests run
8.18. At least 2 times passed integration tests in each environment instance
8.19. Deployment to the test environment

8.20. Check that all the files in the deployment package are the same as those in the latest commit of the dev git branch.
8.21. restart the application layer
9. KNOWN ISSUES AND WORKAROUNDS
9.1. Morbo is stuck
9.1.1. Morbo is stuck
9.1.2. Probable root cause
9.1.3. Known solution and workaround

MAINTENANCE 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. 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.

1.2. 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.

2. COMMON OPERATIONS

2.1. To check the status of the postgreSql

To check the status of the postgreSql issue:

```
sudo /etc/init.d/postgresql status
```

2.2. To stop the postgreSql

To stop the postgreSql issues:

```
sudo /etc/init.d/postgresql stop
```

2.3. To start the postgreSql

To start the postgreSql issues:

```
sudo /etc/init.d/postgresql start
```

2.4. to check the port on which it is listening

To check the port on which it is listening issue:

```
sudo netstat -tulntp | grep -i postgres
# tcp        0      0 127.0.0.1:5432          0.0.0.0:*
LISTEN      8095/postgres
```

2.5. Check the postgres status

Check the postgres status.
Check the port to which the Postgres is running with this command:

```
sudo /etc/init.d/postgresql status

# restart if needed
sudo /etc/init.d/postgresql restart

# check on which ports it is runnning
sudo netstat -plunt |grep postgres
```

2.6. Application Layer run-state management

Remember to cd to the product instance dir you are going to work on for example:
cd /vagrant/opt/csitea/qto/qto.0.6.5.dev.ysg
All the examples bellow are assuming you've done that in advance.

2.6.1. start the application layer

To start the application layer in development mode use the morbo command (debug output will be shown) , to start it in production mode use the hypnotoad pattern

```
# start hypnotoad ( does the stop as well )
bash src/bash/qto/qto.sh -a mojo-hypnotoad-start
# start morbo
bash src/bash/qto/qto.sh -a mojo-morbo-start
```

2.6.2. stop the application layer

To stop the application layer in development mode use the morbo command (debug output will be shown) , to start it in production mode use the hypnotoad pattern. Note that the morbo command does not stop any running morbo on OTHER product instance dir, but the hypnotoad does stop all - aka hypntotoad as the binary of production must be running only on 1 and only one instance on a host.

```
# only stop hypno
bash src/bash/qto/qto.sh -a mojo-hypnotoad-stop
# only stop morbo
bash src/bash/qto/qto.sh -a mojo-morbo-stop
```

2.6.3. Restart OS network service

Sometimes you might just need to restart the whole network service on Ubuntu:

```
sudo /etc/init.d/networking restart
```

2.7. Security related operations

There are 2 security modes of operations in qto:
 - none authenticative one (no login , all can be changed by anyone)
 - native authentication mode - the user credentials are stored per db

2.7.1. Add, modify and delete new users to the application

You as the owner of the instance you are running must be aware that the requests to register to the instances you are operating will come via e-mail. Simply add, update and delete users in the users table and sent the password with prompt to edit it to the new user.

2.7.2. Regular users visibility

Use the following http password generator:
http://www.htaccesstools.com/htpasswd-generator/

3. BACKUP AND RESTORE

You could easily add those commands to your crontab for scheduled execution - remember to add the absolution patch of the qto.sh entry script.

3.1. Load database connectivity configuration securely

Qto provides you with the means and tools to work on tens of databases, yet one at the time. Thus once you open a shell to run the tools you must have the connectivity to the database you want to work on.

```
 # source the db configuration setting func
source lib/bash/funcs/export-json-section-vars.sh

# load the product instance configuration details by calling
that func
doExportJsonSectionVars cnf/env/tst.env.json '.env.db'

# set the psql alias having all the connectivity details for
the instance db
alias psql="PGPASSWORD=${postgres_db_useradmin_pw:-} psql -v
-t -X -w -U ${postgres_db_useradmin:-} --port
$postgres_db_port --host $postgres_db_host"

# now you can run any psql
psql -d my_db -c "\dl"
```

3.2. Backup a database

You backup a database (all the objects, roles and data) with the following one-liner.

```
# obs you must have the shell vars pre-loaded !!! Note dev,
tst or prd instances !
# clear; doParseCnfEnvVars cnf/qto.prd.host-name.cnf
bash src/bash/qto/qto.sh -a backup-postgres-db
```

3.3. 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 !!!
# clear; doParseCnfEnvVars <<path-to-cnf-file>>
bash src/bash/qto/qto.sh -a backup-postgres-table -t
my_table
```

3.4. Restore a database

You restore a database by first running the pgsql scripts of the project database and than restoring the insert data

```
# obs you have to have the shell vars pre-loaded !!!
# clear; doParseCnfEnvVars <<path-to-cnf-file>>

bash src/bash/qto/qto.sh -a run-qto-db-ddl
psql

psql -d $postgres_db_name <
dat/mix/sql/pgsql/dbdumps/dev_qto/dev_qto.20180813_202202.in
srt.dmp.sql
```

3.5. Restore a database table

You restore a database table by first running the pgsql scripts of the project database or ONLY for that table and than restoring the insert data from the table insert file.

```
# obs you have to have the shell vars pre-loaded !!!
# re-apply the table ddl
psql -d $postgres_db_name < src/sql/pgsql/dev_qto/13.create-
table-requirements.sql

psql -d $postgres_db_name < dat/tmp/requirements.data.sql
```

4. USAGE SCENARIOS

4.1. 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.

Verify the inserted data from the db as follows:

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

4.2. Verify the inserted data from the db

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

4.3. 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 ;
```

5. NAMING CONVENTIONS

5.1. 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 ..

5.2. 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

6. 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.

6.1. 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...

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.

```
# 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
```

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

6.3. Zero tolerance for bugs

As soon as bugs are identified and reproduceable, register them as issues and resolve them with prior 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.

6.4. 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!

6.5. Branch for 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.

6.6. Testing and integrations in the tst branch

The tst branch is dedicated for 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
```

6.7. Quality assurance in the qas branch

At this phase all the tests with all the expected functionalities should work at once. No small hotfixes are allowed - if a need arises new branch is created to the tst branch The quality assurance

6.8. 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.

7. 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

7.1. 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.

7.2. 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.

7.3. Chat / IRC

Should you want a quicker respond than 2 hours use thre chat tool

7.4. Documentation

Undocumented feature is not a feature.

7.5. Issues management

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

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.

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

```
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
;
"
```

8. FEATURE IMPLEMENTATION WORKFLOW

8.1. Issue creation

8.2. 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>>

8.3. Requirements creation

Define a formal requirement as soon as possible.

8.4. Problem registration

Problems are usually entities which last for longer time period.

8.5. feature branch creation

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

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

8.6. Create a test-entry point

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

8.7. 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.
Try however to use the same naming convention, and implement with future integrations within the end truly dynamic code.

8.8. 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.

8.9. Unit and / or integration test creation

8.10. Implementation

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

8.11. 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
```

8.12. 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).

8.13. 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.

8.14. DoD check-list walkthrough

Perform the DoD checklist as follows.

8.15. 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.

8.16. 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.

8.17. 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
```

8.18. 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
```

8.19. 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>>
```

8.20. 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:

```
# 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>>
```

8.21. restart the application layer

Well just chain the both commands.

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

9. KNOWN ISSUES AND WORKAROUNDS

9.1. Morbo is stuck

9.1.1. Morbo is stuck

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] $
```

9.1.2. 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] $
```

9.1.3. Known solution and workaround

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>>
```