Table of Contents

ISSUE-TRACKER DEVOPS GUIDE

1. GUIDING PRINCIPLE'S

This section might seem too philosophical for a start, yet all the development in the issue-tracker 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. It should just work

Any instance of the issue-tracker 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.

1.1.1. Personal responsibility

Any given instance of the issue-tracker should have ONE and only ONE person which is responsible at the end for the functioning of THIS instance - so think carefully before attempting to take ownership for 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, nor even promise to help.

1.1.2. 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 behavior when proper testing is implemented.
Testing ensures the consistency and future expandability of the functionalities.

1.2. 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 ...

1.3. Be user-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 artifacts.

2. MAINTENANCE AND OPERATIONS

2.1. 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.

2.2. 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 standars for work deliverables. Either bring it up to the standard level or get rid of it.

2.3. RDBMS Run-state management

2.3.1. To check the status of the postgreSql

To check the status of the postgreSql issue:

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

2.3.2. To stop the

postgreSql

To stop the postgreSql issues:

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

2.3.3. To start the postgreSql

To start the postgreSql issues:

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

2.3.4. 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
```

2.3.5. Check the postgres status

Check the postgres status.
Check the port to which the postres 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
```

2.4. Application Layer run-state management

2.4.1. 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

```
bash src/bash/issue-tracker/issue-tracker.sh -a mojo-hypnotoad-start

bash src/bash/issue-tracker/issue-tracker.sh -a mojo-morbo-start
```

2.4.2. 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

```
bash src/bash/issue-tracker/issue-tracker.sh -a mojo-hypnotoad-stop

bash src/bash/issue-tracker/issue-tracker.sh -a mojo-morbo-stop
```

2.4.3. Restart OS network service

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

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

3. BACKUP AND RESTORE

3.1. Backup a database

You backup a database 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/issue-tracker/issue-tracker.sh -a backup-postgres-db
```

3.2. 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/issue-tracker/issue-tracker.sh -a run-pgsql-scripts
psql

psql -d $postgres_db_name < dat/mix/sql/pgsql/dbdumps/dev_issue_tracker/dev_issue_tracker.20180813_202202.insrt.dmp.sql
```

4. USAGE SCENARIOS

4.1. Shell based actions usage

4.1.1. 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.

```
bash src/bash/issue-tracker/issue-tracker.sh -a increase-date
```

4.1.2. Run xls-to-db action

You insert the date of the daily , weekly , monthly or yearly issues from the daily input excel file(s) by running the xls-to-db action.
If you have the guid column with uuid's than this will be upsert and not bare insert.
You should be able to update only non-nullable column by reducing the number of columns in your xls sheet.

```
export do_truncate_tables=1 ;
bash src/bash/issue-tracker/issue-tracker.sh -a xls-to-db
```

4.1.3. Run db-to-txt action

The db-to-txt action converts your db tables into txt files by using "smart" formatting rules. This feature is deprecated and should work only for tables having the same attributes set as the "issues" tables.

```
bash src/bash/issue-tracker/issue-tracker.sh -a db-to-txt
```

4.1.4. 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/issue-tracker/issue-tracker.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/issue-tracker/issue-tracker.sh -a xls-to-db -a db-to-txt ; done ;
```

4.1.5. Run the issue-tracker file to db load

Run the issue-tracker file to db load

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

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

4.1.6. 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_issue_tracker
```

4.2. web based routes usage

4.2.1. Run the http://<<web-host>>:<<web-port>>/<<proj-db>>/get/<<table>>/<<guid>> route

Load a table with guid's.
Check a single item with your browser, for example:
http://doc-pub-host:3000/dev_stockit_issues/get/companies/727cf807-c9f1-446b-a7fc-65f9dc53ed2d

```
# load the items
while read -r f; do
export xls_file=$f;
bash src/bash/issue-tracker/issue-tracker.sh -a xls-to-db  ;
done < <(find $proj_txt_dir -type f)

# verify the data
psql -d $db_name -c "SELECT * FROM company_eps "
```

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

6.1. 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.

6.2. 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.

6.3. Feature development in a feature branch

You start the development in your own feature branch named : dev--<<issue-id>>-<<short-and-descriptive-name>>.

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.

```
export issue_tracker_project=""; bash src/bash/issue-tracker/issue-tracker.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. SCENARIOS

7.1. A small team project hours tracking scenario

This scenario describes the steps and processes, which could be implemented to achieve a small team (3-10) members issue-tracking with hours reporting by using the issue-handler combined with Google Cloud authentication and storage.

8. WAY OF WORKING

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

8.1. 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.

8.2. 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.

8.3. Chat / IRC

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

8.4. Documentation

Undocumented feature is not a feature.

9. 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.

9.1. 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 issue-tracker 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

9.2. 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>>

9.3. Requirements creation

Define a formal requirement as soon as possible.

9.4. Problem registration

Problems are usually entities which last for longer time period.

9.5. 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
```

9.6. 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.

9.7. Implementation of Proof of Concept

9.8. Prototype implementation

9.9. Unit and / or integration test creation

9.10. Implementation

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

9.11. Deployment and test to the test environment

Deploy to the test environment.

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

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

9.12. 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)

9.13. 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

9.14. DoD check-list walkthrough

Perform the DoD checklist as follows

9.14.1. 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.

9.14.2. 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.

9.14.3. At least 2 times passed unit tests run 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.

9.14.4. At least 2 times passed integration tests run 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.

9.14.5. Deployment to the test environment

Deploy to the test environment as follows:

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

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

9.14.6. 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/issue-tracker/issue-tracker.sh -a to-tst

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

9.14.7. restart the application layer

Well just chain the both commands.

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

10. SYSTEM ADMINISTRATION AND MAINTENANCE OPERATIONS

10.1. Known issues and workarounds

10.2. 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 [issue-tracker][@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/issue_tracker
tcp        0      0 0.0.0.0:3002            0.0.0.0:*               LISTEN      7626/issue_tracker
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 [issue-tracker][@host-name] [4426] STOP FOR issue-tracker RUN with:
 [INFO ] 2018.09.14-10:23:16 EEST [issue-tracker][@host-name] [4426] STOP FOR issue-tracker RUN: 0 0 # = STOP MAIN = issue-tracker
issue-tracker-dev ysg@host-name [Fri Sep 14 10:23:16] [/vagrant/opt/csitea/issue-tracker/issue-tracker.0.4.9.dev.ysg] $
```

10.2.1. 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 [issue-tracker][@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/issue_tracker
tcp        0      0 0.0.0.0:3002            0.0.0.0:*               LISTEN      7626/issue_tracker
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 [issue-tracker][@host-name] [4426] STOP FOR issue-tracker RUN with:
 [INFO ] 2018.09.14-10:23:16 EEST [issue-tracker][@host-name] [4426] STOP FOR issue-tracker RUN: 0 0 # = STOP MAIN = issue-tracker
issue-tracker-dev ysg@host-name [Fri Sep 14 10:23:16] [/vagrant/opt/csitea/issue-tracker/issue-tracker.0.4.9.dev.ysg] $
```

10.2.2. 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>>
```