Moorhouse SQL Developers Process Documentation

Contents

[Monday.com ticketing 2](#_Toc190869060)

[Workflow and statuses 2](#_Toc190869061)

[Additional project boards 3](#_Toc190869062)

[Working on developments 5](#_Toc190869063)

[Ticket folder 5](#_Toc190869064)

[SQL developments 6](#_Toc190869065)

[Saving files 6](#_Toc190869066)

[SQL style guide 6](#_Toc190869067)

[Header and change log 6](#_Toc190869068)

[Performance and wider considerations 8](#_Toc190869069)

[SQL Agent and scheduled jobs 8](#_Toc190869070)

[SSRS report developments 9](#_Toc190869071)

[Approach 9](#_Toc190869072)

[Data sources 9](#_Toc190869073)

[Report Server folders 9](#_Toc190869074)

[Standard formatting 10](#_Toc190869075)

[ETL developments 10](#_Toc190869076)

[Approach 10](#_Toc190869077)

[SSIS deployment 11](#_Toc190869078)

[PowerShell deployment 11](#_Toc190869079)

[Git version control 12](#_Toc190869080)

[Committing changes 12](#_Toc190869081)

[SSRS report projects 13](#_Toc190869082)

[Timesheet 13](#_Toc190869083)

[Requesting annual leave 14](#_Toc190869084)

[CPD recording 15](#_Toc190869085)

# Monday.com ticketing

All work should be ticketed via Monday.com boards. The main board for SQL developers is the **Product – Data – Reports** board located at:

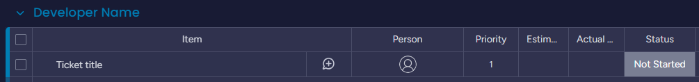
<https://constructaquote.monday.com/boards/127927377>

Additional separate boards may be created for larger projects (see [Additional project boards](#_Additional_project_boards) section below).

Each developer has a group on this board for their priority tickets. Managers or team leaders will add new tickets to these groups and assign them, prompting a notification to the developer within Monday.com and via email.

## Workflow and statuses

A newly created ticket will appear with the status ‘Not Started’:



Tickets should be worked on in the priority that has been assigned by the order of the tickets in your group (highest priority at the top).

The workflow should be as follows (not all steps will apply to all tickets):

|  |  |
| --- | --- |
|  | Set the status to ‘Working on it’ when starting to work on the ticket, and enter the date in the Date Started column. Then begin working on the solution. Work should be saved in a separate folder for each ticket within the SQLDev Git repository (see the [Working on a BAU ticket](#_Ticket_folder_and) section below).  Add new replies or updates to the ticket to describe the changes made, scripts created, etc. There must be enough information here for another developer to release the ticket if reassigned, or to establish what changes were made if problems related to the ticket are raised at a later date. |
|  | If you require more information than has been provided, or an issue prevents proceeding with the ticket, add a new reply or update and tag the person who originally raised the issue by typing ‘@’ followed by their name. Set the status to ‘Awaiting Info’ or ‘Issue’ accordingly. |
|  | When you believe the development work has been completed, set the ticket status to ‘Testing’. Add a new reply or update on the ticket. Type ‘@’ followed by the name of the person who needs to test the ticket to tag and notify them.  For simple changes where you are confident in the solution, you can tag the person who originally raised the issue (usually the Operations team). More complex changes should be tested by the development team’s tester first. |
|  | If the development work has been scripted but you are unable to release the work to the UAT environment (e.g. due to a prior change being tested), set the status to ‘Awaiting UAT release’. |
|  | If testing has been completed by the development team tester, tag the person who originally raised the issue and ask them to provide final sign off for the change to be released to the live environment. Set the status to ‘Awaiting sign off’. |
|  | If sign off has been provided, the change can be released to the live environment. However, changes that take some time to release (e.g. Product and Scheme changes with screen design changes and multiple scripts to run) should be run outside of business hours so as not to impact users.  Additionally, changes to functionality should not be run late in the afternoon or on Fridays, as any unforeseen issues will require the change to be rolled back and investigated and could require multiple team members to work into the evening or over a weekend.  If waiting to release the change at a better time, set the status to ‘Awaiting Release’. |
|  | After releasing the changes to functionality, tag the development team tester and ask them to perform a sanity check to confirm the change is working as expected in live. For small changes, reports, etc., this step can be skipped.  Be sure to clean up anything that is no longer needed after live release. For example, test versions of reports on the Report Server, test versions of stored procedures in the live database, etc. |
|  | When the ticket is complete, released to live, and confirmed as working, and files are committed to Git (see [Git version control](#_Git_version_control) section below), change the status to ‘Done’. It can then be moved to the board’s completed tickets section for the current month. |

## Additional project boards

Larger projects such as Product and Scheme changes will require a separate board to manage the project. The board may be created by a manager or by the developer. The statuses may differ from those shown above, depending on the nature of the project.

A board can be created with a copy of the structure of an existing board by clicking on the menu button at the top right of the existing board and then selecting **More actions > Duplicate board** as shown:

A screenshot of a computer

Description automatically generated

The next window will give the option to duplicate the structure only, without copying all the tickets.

The items on a project board should be organised into groups. Some examples are as follows, although the exact groups will depend on the project:

* **Requirements:** The requirements document received from the business, and any subsequent versions, should be upload to an item in this group.
* **Questions:** Create separate items for each question arising from the requirements that needs clarification.
* **Specification:** Once the developer has understood the requirements, a technical specification should be produced, detailing the steps that will be taken. This can be uploaded here.
* **Development tasks:** Create an item for each development task that needs to be actioned, for example screen changes, database scripts, etc. Add the estimated time in hours to a number column. The group will then show the total estimated time to complete the development.
* **Development issues:** Issues that arise during the development should be logged here.
* **UAT deployment:** Create items for each step required to release the changes to the UAT environment. Add updates to the items with full instructions so that any developer can perform the release should the lead developer be unavailable.
* **Functional testing:** The development team tester will add questions and issues here that arise during testing.
* **Operational testing:** The operations team will add questions and issues here that arise during testing.
* **Pre live changes:** Changes or additional requirements raised after the development tasks have been estimated should be added here. It’s important to enter the actual time taken on each item as this will be over and above the estimated time for the development.
* **Deployment:** Create items for each step required to release the changes to the live environment. This will form the release plan. Add updates to the items with full instructions so that any developer can perform the release should the lead developer be unavailable. Steps may be broken down into subitems, or bullet points / checklists in an update (see sample image below).
* **Post live issues:** Any issues that are raised after the live deployment should be added to this group.
* **Post live changes:** Any changes that are requested after the live deployment should be added to this group.

An example of a release plan, utilising a checklist to document the steps in each item:

A screenshot of a video game

Description automatically generated

# Working on developments

## Ticket folder

Except for very small changes, BAU (business as usual) tickets will most likely require files such as documentation and scripts to be saved.

Create a new folder for the ticket as a subfolder of the yearly tickets folder within the SQLDev Git repository (see [Git version control](#_Git_version_control) section below). Name the folder with the ticket number and a short description. For example:

Git\SQLDev\Tickets\2005\8142413832 UK Postcode Referral

The ticket number can be copied from the Pulse ID column on the board, or by clicking on the ticket and copying the ID from the browser address bar.

Files should be organised into the following subfolders if applicable:

* **Documentation:** Save any files attached to the original ticket here (requirements, example data, etc.)
* **Originals:** This folder must contain scripts to restore the original versions of any objects changed (stored procedure scripts, etc.) so that the change can be quickly reverted in the event of unforeseen issues.
* **Release:** This folder must contain all scripts and files that form part of the live release.

Be sure to commit the changes to Git and push them to Github.com (see [Git version control](#_Git_version_control) section below) at regular intervals to prevent loss of files that are only saved locally.

## SQL developments

A SQL development will usually require a change to a database object such as a stored procedure. As a first step, the live procedure should be compared against the version in UAT to make sure there are no outstanding unreleased changes from another ticket. Larger scripts can be compared using a differencing and merging application like WinMerge.

### Saving files

Release script filenames for SQL database objects should conform to the following pattern:

[Schema].[ObjectName]

For example: [dbo].[MCLIAB\_uspCalculator]

Following this naming convention will prevent duplicate scripts for the same object from being added to Git version control (see [Git version control](#_Git_version_control) section below).

### SQL style guide

While each developer will have their own preferences and style for formatting SQL scripts, they should aim for consistency with existing Moorhouse-produced code. This will improve readability so that other team members can understand the code, and make it easier to use differencing and merging tools, etc.

Standard formatting employed in Moorhouse scripts includes, but is not limited to:

* Upper case for reserved keywords (SELECT, FROM, etc.)
* Tables prefixed with scheme name (e.g ‘dbo’)
* Square brackets around all schema, table, field and other object names
* Aliases used for all joined tables, and all fields prefixed with the alias
* All keywords, fields and joins on a new line, and indented for clarity
* Commas at start of lines in lists instead of at the end

An example script is shown below:

A close-up of a computer code

Description automatically generated

As shown in the above example, comments should be added to explain the use of IDs and any complex logic to aid other developers in quickly understanding why the script does what it does.

### Header and change log

Scripts for database objects such as stored procedures and triggers should include a header showing the author, date created, and a short description of what the script does.

Following the header, a commented change log should be kept up to date, showing the changes made to the object, who made them, and a short description of the change, including ticket number if applicable.

A commented “test harness” should also be included that sets test values for any input parameters and allows quick testing of the script.

An example script is shown below:

A screenshot of a computer program

Description automatically generated  
Synonyms

When referencing objects in a remote databases via linked servers, synonyms should be created in the calling database, pointing to these objects. This simplifies the changes required if the server name changes, since the server prefix only needs to be changed in the synonym instead of searching for all objects that reference it.

However, synonyms can be confusing as they make an object appear to exist locally when it doesn’t. Therefore, they should be used minimally. When referring to tables in a different database on the same server, the preference is to prefix with the database name rather than utilizing a synonym.

### Performance and wider considerations

A large proportion of our SQL developments will read directly from the live Transactor\_Live database. This heavily used database supports the TGSL TCAS application, Constructaquote and XBroker web sites, numerous SSRS reports, and data imports and exports.

It is crucial that all released code performs well and does not cause table locking or block other database processes which could prevent quotes from running and policies being sold.

Some considerations and suggestions:

* Begin all scripts by setting the transaction isolation level to read uncommitted transactions as follows, as this will prevent tables being locked:  
  SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
  While this is generally frowned upon by the SQL community due to the likelihood of “dirty reads”, we have not encountered any major issues with using it.
* Selecting data from tables is generally more performant than using views, especially if the views contain many other columns and joins that are not needed for the solution.
* Table variables are generally faster than temporary tables as they are held in memory, whereas temporary tables are written to disk (but if the table variable becomes too large it may spill over and write to disk anyway).
* While we want to keep scripts simple to read, sometimes it is better to break code down into multiple steps rather than trying to do everything in a single statement. For example, an update statement with lots of joins, CTEs (common table expressions), etc. may work, but as the data changes over time, the query engine changes the execution plan and performance is suddenly degraded. It may be better to create a multi-stage programme, for example inserting initial records into a table variable or temporary table, and then having subsequent update statements to add data from other tables. This way the developer has control over the order of operations instead of relying on the query engine to decide.
* Is any code running unnecessarily? For example, if amending a Scheme calculator to refer, does it still need to run all the code to calculate a premium? Preventing this from running may only save a fraction of a second but this can add up when multiplied over several Schemes. We want the customer to obtain a quote in the fastest time possible.

### SQL Agent and scheduled jobs

When developing a script that needs to be scheduled to run on a regular basis, the preference is to create a stored procedure rather than hard coding the script directly into a SQL Agent job window. The reason for this is that the content of stored procedures is searchable by Redgate SQL Search and any objects (tables, views, etc.) will be shown when viewing dependencies on an object in SQL Server Management Studio. These dependencies are not reported if contained in a SQL Agent job script.

Scheduled jobs must contain error handling to roll back any updates if an error occurs. Otherwise, the job may partially run and cannot be easily re-run without updating certain records for a second time, for example.

Scheduled jobs should not rely on the system date to identify records that need to be updated, as the job may need to be re-run at a later date if an error occurs. It is best practise to store the last run date in a table upon successful completion of a job. While various tables have been used historically, the agreed table to use is the TGSL\_Job\_Log table in the Transactor\_Support database. The next time the procedure runs it can select records based on the last run date.

## SSRS report developments

### Approach

The SQL development team has chosen to use Visual Studio as the report development environment. Report RDL files should not be edited outside of the Visual Studio report projects as this can lead to changes being overwritten by other developers.

Similarly to SQL Agent jobs, when developing an SSRS report, the preference is to create a stored procedure to return the main data to the report rather than hard coding the script directly into the query window in Visual Studio. Again, the reason for this is that the content of stored procedures is searchable by Redgate SQL Search and any objects (tables, views, etc.) will be shown when viewing dependencies on an object in SQL Server Management Studio.

Smaller queries can be hard coded in the report, for example selecting a list of Agents to populate a parameter drop-down.

### Data sources

The Visual Studio projects contain shared data source connections to databases on the UAT server (MHGSQL01\TGSLTEST) as reports should be developed against test data where possible, to avoid impacting live systems.

However, Operations and users will often want to test report changes against live data. If making a change to an existing report, a copy of the report procedure can be created in the live database with a suffix of “\_test” and a copy of the report definition file uploaded which has been edited in a text editor to point to the test version of the stored procedure.

Any test versions of procedures and reports should be deleted after the live release to reduce clutter.

### Report Server folders

The main TGSL Report Server (accessible at <http://mgl/Reports/browse/>) contains the following main folders into which reports should be deployed accordingly:

* **Archive:** Old reports we believe are no longer needed (they can be deleted if execution logs show they haven’t been accessed for a number of years)
* **Bordereaux:** Live bordereaux reports (lists of sold policies that get sent to the insurers)
* **Bordereaux/UAT:** Test versions of bordereaux reports pointing at the UAT database
* **DBA:** Reports about Report Server usage, etc., available to a limited number of users
* **Reports:** The main live reports folder, available to most users
* **Reports/UAT:** Test versions of reports pointing at the UAT database
* **Reports – Test:** Test versions of live reports pointing at the live database
* **Restricted:** Reports that act as an interface to search or append data that are only available to a limited number of users
* **Scripts:** Reports that act as an interface to call update procedures that update data in the live database; highly restricted to users who need them and understand how to use them

All test reports should be deleted from the Test and UAT folders after the live release to reduce clutter.

### Standard formatting

The design of new reports should conform to the standard formatting provided by the MI team, as shown in this example:

A screenshot of a report

Description automatically generated

The typeface is Arial and the colours are as follows:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## ETL developments

### Approach

A number of extract, transform and load (ETL) processes exist to extract data and send to insurers and other third parties, for example by uploading to FTP servers. There are also processes to download files from third parties such as credit providers, data enhancement services, etc.

Historically, most ETL processes have been created as SSIS packages. However, more recently PowerShell has been used due to the ease of creating and updating scripts. The approach taken to any new requests is at the discretion of the developer, but should take into account the following considerations:

* How easy will it be to make future changes?
* Can the dependencies required for data extract be viewed in SQL Server Management Studio, i.e. through the use of stored procedures?
* If a large amount of data is being extracted, can the timing of the process be compared alongside other jobs, i.e. by extracting the timings of SQL Agent jobs?

### SSIS deployment

SSIS packages are deployed on the data warehouse server MHGDW01 using the SSIS Catalog deployment method. The main catalog is called SSISDB:

MHGDWOI (SQL server 1 5.0.2D80.g - MOORHOUSE\jeremai.sr 
Databases 
Security 
Server Objects 
Replication 
Poly8ase 
Always On High Availability 
Management 
Integration Services Catalogs 
SSISD8 
Compare_The Market 
Confused 
Crusader 
Fleet 
Sanctions List 
• TGSL 

### PowerShell deployment

PowerShell scripts are saved in the following location on the data warehouse server MHGDW01:

\\MHGDW01\s$\PowerShell

The scripts are scheduled to run at certain times using the Windows Task Scheduler on MHGDW01. Existing scheduled scripts can be found in the root Task Scheduler Library folder:

A screenshot of a computer

AI-generated content may be incorrect.

When setting up a new task it is recommended to set the following option in the settings, unless there is a time sensitivity that means the task must run at exactly the scheduled time:

A screenshot of a computer

AI-generated content may be incorrect.

# Git version control

For the purposes of this document, it is assumed that the developer already has Git set up and repositories cloned. If not, see separate documentation for setting up Git.

The SQL development team primarily uses Git for version control of files, allowing older versions of database scripts, for example, to be reviewed if problems arise after changes have been made.

This means most work is done on the Master branch of the SQLDev repository. For more complex development there may be cases where branching and merging is used.

## Committing changes

After releasing work to live environments, files should be committed to Git and pushed to Github.com. The overall structure of the folders in the SQLDev repository is as follows (additional subfolders may be created as needed):

SQLDev

* 🗁 Databases
  + 🗁 <Database name>
    - 🗁 SQL scripts
* 🗁 Documentation
  + 🗁 <Documentation type>
* 🗁 PowerShell scripts
  + 🗁 Archive
  + 🗁 Asset
  + 🗁 Connections
  + 🗁 Functions
  + 🗁 Toolbox
* 🗁 Products
  + 🗁 <Product name>
    - 🗁 Schemes
      * 🗁 <Scheme name>
        + Documentation
        + WPD file
    - 🗁 Screen Designer project
* 🗁 SSRS Reports
  + 🗁 <Visual Studio solution name>
    - 🗁 <Deployment folder name>
* 🗁 SSIS Packages
  + 🗁 <Package name>
* 🗁 Tickets
  + 🗁 <Year>
    - 🗁 <Ticket number> <Description>

Copy release files from the ticket folder to the relevant folders and then commit via Git commands or a Git client such as GitHub desktop. The summary and description should include the ticket number and a link back to the ticket on Monday.com, similar to the following example:

A screenshot of a computer

Description automatically generated

## SSRS report projects

Note that after working on SSRS projects in Visual Studio, Git will identify a number of new files. Some of these do not need to be committed.

Any files in the following subfolders of any report projects can be discarded:  
\FileContentIndex  
\bin\Debug

If using GitHub Desktop, simply right-click and choose **Discard changes**:

A screenshot of a computer

Description automatically generated

# Timesheet

Each developer has a separate Monday board to use as their personal timesheet. This is only visible to managers and / or team leaders.

It is advisable to complete the time sheet on a daily basis, but it must be completed by the end of each week. As insurance is a regulated industry, we are required to account for all of our time.

Create a new group for each week on the timesheet and fill in the columns as described below and shown in the example:

* **Item:** A short description of the work, e.g. title of the ticket worked on
* **Date:** If the same item is worked on over multiple days, it is advisable to split into daily entries with comments about what was achieved each day. This can be valuable should issues arise later and we need to see the exact date that a change was made, for example.
* **Status:**
  + ‘Bank Holiday’ and ‘Annual Leave’ as applicable
  + ‘Admin’ for the daily standup call, writing documentation, meetings, etc.
  + ‘BAU’ (business as usual) for ticketed support work, fixing errors, making small changes, etc.
  + ‘Project’ for larger project work such as new Products, Schemes, reports, etc. Ask if unsure, as project time is charged to a different part of the organisation.
* **Hours:** Must add up to 37.5 hours for the week
* **Comments:** Enter a short description of the work undertaken
* **Link to ticket:** Copy and paste a link to the ticket (if applicable)

A screenshot of a computer

Description automatically generated

# Requesting annual leave

Please ensure that:

* Annual leave requests are submitted with a reasonable notice period where possible
* Annual leave is taken throughout the year to avoid a situation where all team members are using up leave at the end of the year (note that carrying over annual leave to the following year is not permitted)

Annual leave is requested through the Breathe HR web interface, accessible both internally and externally at the following location:

<https://login.breathehr.com/login?redirect_app=hr>

After logging in, click the calendar icon at top right, then select ‘IT’ from team drop-down to view the team calendar:

A screenshot of a computer

Description automatically generated

If another SQL developer has requested the dates you wish to request please check with a team leader or manager.

# CPD recording

As we work in a regulated industry, it is a legal requirement that each developer completes 15 hours of continuous professional development (CPD) every year. This includes online training modules that get assigned, plus self-paced learning such as searching the internet for solutions to assist with complex developments.

In addition, SQL developers should try to regularly read SQL web sites and blogs to learn about new features and tips and tricks. This can also be recorded as CPD time.

CPD is recorded by logging into the Aviva Development Zone at <https://www.developmentzone.co.uk/>. The home page dashboard shows a graph of the number of hours completed towards the 15 hour target. To record more CPD, click on **Record CPD** from the top menu, then click the **Add CPD Activity** button and fill out the form.