

**SnapLogic Integration Guidebook**

**Standards and guidelines**

Version 1.0

Table of Contents

Contents

[**1. Document control 8**](#_heading=h.2jxsxqh)

[**2. Introduction 9**](#_heading=h.206ipza)

[**3. Scope 9**](#_heading=h.1egqt2p)

[**4. Conventions 10**](#_heading=h.1jlao46)

[4.1. Document Conventions 10](#_heading=h.43ky6rz)

[4.2. Code Conventions 10](#_heading=h.2iq8gzs)

[4.2.1. Why have code conventions 10](#_heading=h.xvir7l)

[4.2.2. Adherence and Enforcement 10](#_heading=h.3hv69ve)

[4.3. Naming Conventions 11](#_heading=h.1x0gk37)

[4.3.1. Project spaces 11](#_heading=h.4h042r0)

[4.3.2. Projects 11](#_heading=h.2w5ecyt)

[4.3.3. Pipelines 12](#_heading=h.1baon6m)

[4.3.4. Pipeline parameters and fields 14](#_heading=h.3vac5uf)

[4.3.5. Triggered and Ultra tasks 14](#_heading=h.2afmg28)

[4.3.6. Accounts 15](#_heading=h.pkwqa1)

[4.3.7. Snaps 16](#_heading=h.39kk8xu)

[**5. Code / Development 17**](#_heading=h.1opuj5n)

[5.1. Versioning 17](#_heading=h.48pi1tg)

[5.1.1. Pipeline versions 17](#_heading=h.2nusc19)

[5.1.2. GitHub versioning 18](#_heading=h.1302m92)

[5.2. Pipeline parameters 22](#_heading=h.3mzq4wv)

[5.3. Clean and optimized code 24](#_heading=h.2250f4o)

[5.4. Expression libraries 25](#_heading=h.haapch)

[5.5. Error handling 26](#_heading=h.319y80a)

[5.5.1. General error handling options 26](#_heading=h.1gf8i83)

[5.5.2 Error message format 27](#_heading=h.40ew0vw)

[5.6. Logging 30](#_heading=h.2fk6b3p)

[5.7. Documenting the code 31](#_heading=h.upglbi)

[**6. Code Migration strategy 34**](#_heading=h.3ep43zb)

[**7. github code maintenance using snaplogic 35**](#_heading=h.1tuee74)

[7.1. Environments 35](#_heading=h.4du1wux)

[7.2. General notes and cautions 35](#_heading=h.45jfvxd)

[7.2.1. General GitHub notes 35](#_heading=h.2koq656)

[7.2.2. General Snaplogic notes 35](#_heading=h.zu0gcz)

[7.3. Workflow diagram 36](#_heading=h.3jtnz0s)

[7.4. Step by step migration guide 36](#_heading=h.1yyy98l)

[7.4.1. Create Snaplogic Projects 36](#_heading=h.4iylrwe)

[7.4.2. Create GitHub repository 37](#_heading=h.2y3w247)

[7.4.3. Create develop branch 38](#_heading=h.1d96cc0)

[7.4.4. Checkout DEV and TEST Snaplogic projects to develop branch 38](#_heading=h.3x8tuzt)

[7.4.5. Commit assets on develop branch 39](#_heading=h.2ce457m)

[7.4.6. Pull from develop branch 40](#_heading=h.rjefff)

[7.4.7. Merge master branch 41](#_heading=h.3bj1y38)

[7.4.8. Create release tag 43](#_heading=h.1qoc8b1)

[7.4.9. Checkout STAGE and PROD on release tag 43](#_heading=h.4anzqyu)

[7.4.10. Delete/Remove assets from SnapLogic and GitHub 45](#_heading=h.14ykbeg)

[**8. Adminstration 46**](#_heading=h.3oy7u29)

[8.1. Managing users and groups 46](#_heading=h.243i4a2)

[**9. Microservices 47**](#_heading=h.j8sehv)

[**10. Child pipelines 47**](#_heading=h.338fx5o)

[**11. Notifications 48**](#_heading=h.1idq7dh)

[11.1. 8.1 Building email notification that contains table 49](#_heading=h.42ddq1a)

[11.2. 8.2 Task and Snaplex notifications 50](#_heading=h.2hio093)

[**12. TRIGGER VS ULTRA TASK 51**](#_heading=h.wnyagw)

List of Figures

[Figure 1 – Example of correct pipeline naming 10](#_heading=h.3gnlt4p)

[Figure 2 – Pipeline name structure 10](#_heading=h.1vsw3ci)

[Figure 3 – Default task name 11](#_heading=h.4i7ojhp)

[Figure 4 – Example of correct task name 11](#_heading=h.1ci93xb)

[Figure 5 – Example of correct task name 11](#_heading=h.1ci93xb)

[Figure 6 – Example of incorrect snap naming 12](#_heading=h.3as4poj)

[Figure 7 – Example of correct snap naming 12](#_heading=h.1pxezwc)

[Figure 8 – Creating pipeline version 13](#_heading=h.3o7alnk)

[Figure 9 – GitHub versioning flow 14](#_heading=h.ihv636)

[Figure 10 – SnapLogic GitHub app 14](#_heading=h.32hioqz)

[Figure 11 – User settings dialog 15](#_heading=h.1hmsyys)

[Figure 12 – GitHub authorization in User settings 15](#_heading=h.41mghml)

[Figure 13 – Checking out GitHub repository 16](#_heading=h.2grqrue)

[Figure 14 – Selecting GitHub branch/tag 16](#_heading=h.vx1227)

[Figure 15 – GitHub tracked project 17](#_heading=h.3fwokq0)

[Figure 16 – GitHub actions 17](#_heading=h.1v1yuxt)

[Figure 17 – GitHub commit info 18](#_heading=h.4f1mdlm)

[Figure 18 – OPEN API selection 19](#_heading=h.19c6y18)

[Figure 19 – Correct solution for account splitting 20](#_heading=h.28h4qwu)

[Figure 20 – Example usage of function from expression library 21](#_heading=h.37m2jsg)

[Figure 21 – Usage of expressions for expression library 21](#_heading=h.1mrcu09)

[Figure 22 – Result from execution of expression from expression library 21](#_heading=h.46r0co2)

[Figure 23 – Screenshot of Expression Library named “mappingPipeName” 21](#_heading=h.2lwamvv)

[Figure 24 – Example of usage of “getPipelineName” from configuration file 22](#_heading=h.111kx3o)

[Figure 25 –Stop pipeline execution behavior 22](#_heading=h.4fsjm0b)

[Figure 26 – Discard error data and continue behavior 22](#_heading=h.4k668n3)

[Figure 27 – Route error data to error view behavior 22](#_heading=h.2zbgiuw)

[Figure 28 – Error object 23](#_heading=h.3ygebqi)

[Figure 29 – Writing error in log file 23](#_heading=h.3ygebqi)

[Figure 30 – Example of log table in database 24](#_heading=h.sqyw64)

[Figure 31 – Example of email message 25](#_heading=h.2r0uhxc)

[Figure 32 – Example of errors summary 25](#_heading=h.1664s55)

[Figure 33 – Building email body in with expressions 26](#_heading=h.3q5sasy)

[Figure 34 – Configuring of email sender snap 26](#_heading=h.25b2l0r)

[Figure 35 – Task notification settings 26](#_heading=h.kgcv8k)

[Figure 36 – Example of pipeline notification email 27](#_heading=h.34g0dwd)

List of Tables

[Table 1 – Document history 7](#_heading=h.3znysh7)

[Table 2 – Pipeline naming 10](#_heading=h.1ksv4uv)

[Table 3 – Naming parameters and fields 10](#_heading=h.3j2qqm3)

[Table 4 – Example account naming 12](#_heading=h.2bn6wsx)

# Document control

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comment |
| 0.1 | 26/12/2019 | Darko Velkov | Draft version |
| 0.2 | 27/12/2019 | Petar Rajchinoski | Added pipeline parameters, fields, triggered and ultra-tasks |
| 0.3 | 23/04/2020 | Mile Grujovski | Document reformatting |
| 0.4 | 18/11/2020 | Igor Perchakovski | Added section 7 |

*Table 1 – Document history*

# Introduction

This document describes the implementation guidelines and best practices for design, construction, administration, configuration and maintenance of solutions in SnapLogic, making it a valuable asset for teams of developers that provides a reference to proven software engineering principles.

# Scope

The scope of this document is to describe the standards and guidelines for producing SnapLogic code (pipelines). These standards are based on sound, proven software engineering principles that lead to code that is easy to understand, to maintain and extend. Besides naming conventions, this document will also describe the best architecture patterns and principles for organizing a new SnapLogic environment. Additionally, following a common set of coding standards leads to greater consistency, making teams of developers more productive. These guidelines should help in creating a uniform coding habits so that reading, checking and maintaining code becomes easier. This document is focused mainly on defining guidelines for SnapLogic integration platform, however guidelines are applicable for other situations also.

# Conventions

## Document Conventions

This document attempts to provide a flexible, usable set of coding conventions. The language used in describing each convention dictates the level of flexibility:

* **Must** – a standard must be followed
* **Should** – a guideline that should be strongly considered
* **Can** – an optional convention whose use is at the discretion of the team or individual developer

All naming and code samples are presented in Courier 11-point font:

This is sample code.

## Code Conventions

### Why have code conventions

Code conventions are important for a number of reasons:

* Majority of the lifetime cost of a software solution goes to maintenance
* Hardly any software is maintained for its whole life by the original author. Code must be easily understood by a maintenance engineer.
* Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly. Easily understood code is more likely to be reused.

If you present your source code as a product, you need to make sure it is as well packaged and clean as any other product you create. Source code is a corporate asset.

### Adherence and Enforcement

These standards are to be adhered to by all project teams throughout the development project. A process of peer code reviews will drive enforcement of these standards and all team leaders will be responsible for ensuring adherence to the standard within their individual projects.

This standard does not address every issue that will arise throughout the course of a project and it is not intended to do so. Every reasonable attempt should be made to adhere to these standards. As the need arises, standards should be changed and extended. When a judgment call is required, go with readability and maintainability first.

## Naming Conventions

### Project spaces

Project spaces are added above projects and represent the highest level of hierarchy in the current offering in SnapLogic. Project spaces are named using pascal case format, for example: **NameOfProjectSpace**.

1. (**Should**) Project space name should reflect the functionality or the problem solution that project is solving, for example: “TransportPlanning”, “DutyAllocation” or “InvoiceApproval”.
2. In some cases, it makes more sense the project space to be named after some system from whom all integrations are related, For example: “Workday”, “NetSuite”, etc.

If the client has only one SnapLogic organization that will be used for development, testing and production than we have slightly different naming convention for project spaces. Both of the above scenarios are still valid only the name of the working environment is added as prefix. Example for scenario 1: “DEV\_ TransportPlanning”, “PROD\_ DutyAllocation ” etc.

(**Should**) Avoid using one SnapLogic organization for different environment. Best practice is using different organization for each environment. Here are some pros of using different organization per environment:

* SnapLogic pipelines are fully separated on each environment
* In case of unintentional overload (memory or CPU) on some environment the other will not be affected
* SnapLogic releases can be made on each organization separately
* Simpler naming convention for the Project spaces

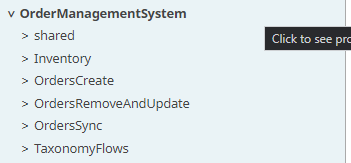
### Projects

Projects represent the second level under the project spaces, it is being used for additional organization of the business logic implemented in SnapLogic. The projects are packages of integrations which can be tested together. It should contain sets integrations which have a parent / child relationship and complete one unique activity. Projects are named by using pascal case format, for example: **NameOfProject**. When selecting a project name consider the following:

**(Can)**

1. Functionality – Example: “Shipment”, “Invoicing”, “Sync”, “DeltaLoad”.
2. Entity – Example: “Product”, “Inventory”, Order, Employee etc.
3. System – Example: “Workday”, “SalesForce”, “LegacySystemName”.
4. Combination of above.

Example of project space and projects:



### Pipelines

A pipeline is a collection of two or more Snaps linked together to orchestrate a flow of data between end points. Correct naming of the pipelines is curtailed and important cause it contributes directly on project maintenance, debugging, migrating and time efficiency especially in big teams as the projects gets more complex and the number of pipelines increase.

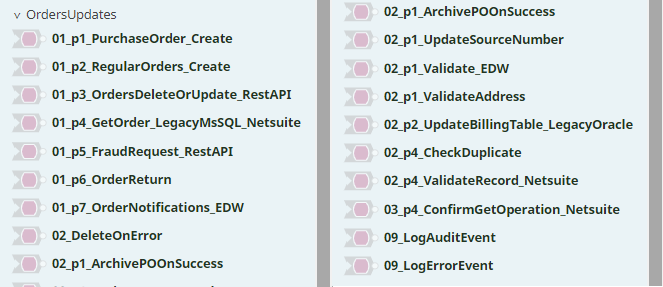
General points that **(should)** be known and followed for naming of pipelines:

* Pipelines in SnapLogic designer are maintained alphabetically.
* **(Should)** Pipelines are named by using pascal case format – **NameOfPipeline**.
* **(Should)** Use “09\_” to name pipelines that are used across multiple pipelines in the same project. Shared business logic (09\_DeleteRecordsOnError) or functionalities (09\_LogAuditEvent, 09\_LogErrorEvent).
* **(Should)** If there are pipelines like error handling that are used by all or a lot of projects they should be placed on a shared level (shared directory) starting with 00\_
  + Example**:** 00\_ErrorHandling

Pipeline naming convention: **[Level]\_[pipelineReference]\_[Entity][OperationAppliedOnEntity]\_[SourceSystem]\_[TargetSystem]**

Example: **01\_p4\_GetOrder\_LegacyMsSQL\_Netsuite**

* **(Must) [Level] –** Numbers only. Starting with 01. Increment for each sub/child pipeline.
* **(Must) [pipelineReference]** - Reference to a parent pipeline. This should be unique for the appropriate process / integration / group of pipelines. See examples below.
  + One or more pipelines can belong to this group and they should have the same reference.
  + Reference should be a short word or abbreviation of few (up to 5-6) characters.
    - Example:
  + If you can’t define short abbreviation or word that will be used as pipelineReference use generic p1, p2, p3 … pn
    - Example: 01\_p4\_...
* **(Must) [Entity][OperationAppliedOnEntity] –** Enter the entity name and operation in pascal case format.
  + Operation is usually a verb.
  + There are some cases when operation can be placed before the entity because it is more logically.
    - Example: 01\_p4\_GetOrder\_LegacyMsSQL\_Netsuite
    - We wrote GetOrder instead of OrderGet but is sounds more logical
* **(Should) [SourceSystem] –** The name of the source system if available
* **(Should) [TargetSystem]** - The name of the target system if available



|  |  |  |  |
| --- | --- | --- | --- |
| Level | PipelineName | Relation | Explanation |
| 1st | 01\_p1\_PurchaseOrder\_Create |  | When level is 01 or Main, it indicates unique identifier of the pipeline and must not be duplicated. |
| 1st | 01\_p2\_RegularOrders\_Create |  | When level is 01 or main, it indicates unique identifier of the pipeline and must not be duplicated. |
| 1st | 01\_p4\_GetOrder\_LegacyMsSQL\_Netsuite |  | When level is 01 or main, it indicates unique identifier of the pipeline and must not be duplicated. |
| 2nd | 02\_p1\_ArchivePOOnSuccess | \_p1\_ | When level is 02 explain that is direct child of  01\_p1\_PurchaseOrder\_Create |
| 2nd | 02\_p1\_UpdateSourceNumber | \_p1\_ | When level is 02 explain that is direct child of  01\_p1\_PurchaseOrder\_Create |
| 2nd | 02\_p4\_CheckDuplicate | \_p4\_ | When level is 02 explain that is direct child of  01\_p4\_GetOrder\_LegacyMsSQL\_Netsuite |
| 3rd | 03\_p4\_ConfirmGetOperation\_Netsuite | \_p4\_ | If lvl is above 02, it explains that is related with  01\_p1\_PurchaseOrder\_Create |
| 2nd | 02\_DeleeteOnError | multiple | When level is present but not pipeline reference it explain it is invoked by multiple parent pipelines |

*Table 2 – Pipeline naming*

### Pipeline parameters and fields

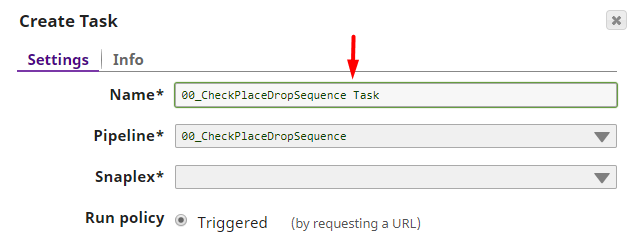
The use of parameters (\_parameters) and fields ($fields) varies depending where they are used within the SnapLogic Elastic Integration Platform and the type of property. Use Camel Case **(Must)** for naming the parameters and fields. The following table explain how they should be named in SnapLogic.

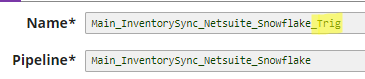
|  |  |  |
| --- | --- | --- |
| Name | Parameter name | Field name |
| Start date | \_startDate | $startDate |
| REST API URL | \_restApiUrl | $restApiUrl |
| Key | \_key | $key |
| Account ID | \_accountId | $accountId |

*Table 3 – Naming parameters and fields*

### Triggered and Ultra tasks

**(Must)** When creating Triggered or Ultra task, by default next to the original name of the pipeline will be added ‘ Task’. Make sure you delete this empty space before ‘Task’. And add “\_Trig” or “\_Ultra” for Trigger and Ultra tasks respectively.



*Figure 3 – Default task name*

*Figure 4 – Example of correct task name*

### Accounts

Accounts can be created in SnapLogic for specific Snap Packs. The name of the Accounts should be meaningful and describe the purpose of the Account. When naming accounts, we suggest to include following information in the name:

1. Type of the account (\_REST\_;\_Gmail\_;\_MSSQL\_;\_Salesforce\_)
2. Account description

Pattern: <TypeOfTheAccount>\_<AccountDescription>

Example: REST\_Weather

Optional:

1. Type of connection: oauth2
2. Name of the project space
3. If there are more organizations (recommended scenario), then the name of the environment in which that specific Account is created **should not be used**. Exception is when we have more than one account for specific
   1. (\_DEV/\_TEST/\_PROD)

Pattern: <TypeOfTheAccount>\_<AccountDescription>\_<TypeOfConnection>\_<ProjectSpaceName>\_<AccountEnvironmentDescription>

Example: REST\_Weather\_OAuth2\_WeatherReporting\_TEST

All parts of the account name should be separated by underscore. In case of database account, a good practice is to include the name of the database as part of the account name.

The table below explains how the Accounts for specific Snap Pack should be named:

|  |  |
| --- | --- |
| Account type | Account name |
| SQL Server | MSSQL\_WeatherDB |
| REST OAuth2 | REST\_Unicorn |
| Gmail | Gmail\_SnapLogicNotification |
| Snowflake | SnowFlake\_InventoryManagementDB |

*Table 4 – Basic account naming examples (without optional extensions)*

(**Should**) Always store accounts in project if they are required to keep connection details related only to that project. If the connection details need to be shared across multiple projects store the account under the project space shared directory.

Avoid storing sensitive accounts directly under the Project Spaces shared directory. This directory is visible to all users of the current organization.

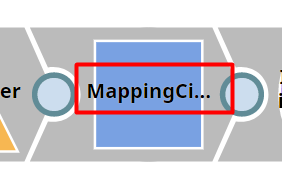
**(Must)** Do not use your personal credential for creating any account that will be used by the projects. Please make sure that, you have service accounts for all of your project that have been migrated to production, as individual user accounts that belong to multiple organizations are subjected to a password expiry, so in between the time the password expired and you have completed the reset, the client will face a failure of service.

### Snaps

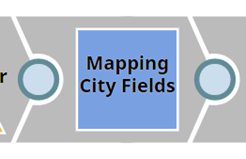
It is a good practice to assign names to the snaps during the pipeline development process. Snap name should be descriptive and represent the functionality that the particular snap is performing.

**(Should)** When name is composed of multiple words then separate each word with empty space. Following words can start with Capital letter but it is suggested to use lower case.

Following examples (Figure 6 and Figure 7) show a Mapper snap that is named by the functionality that it performs. The Mapper snap here is used for mapping fields for a city from source to target. In the example in Figure 6 all words are joined together without empty spaces between them and we cannot see the full name of the snap. On Figure 7 the same snap is named correctly by using empty space between words, so that full name is visible of the snap.



*Figure 6 – Example of incorrect snap naming*



*Figure 7 – Example of correct snap naming*

# Code / Development

## Versioning

### Pipeline versions

This is not classical versioning and is most used while in development process. In order to track the pipeline changes during the development process use the pipeline versioning functionality.

Pipeline versions enables easy replacement of existing pipeline with a newer one, or to roll back changes to a previous version in case when something is wrong. However, note that once you revert to a previouSLy versioned pipeline all your new changes of the current instance of the pipeline will be lost if they have not been versioned.

Figure 8 shows new pipeline version screen.



*Figure 8 – Creating pipeline version*

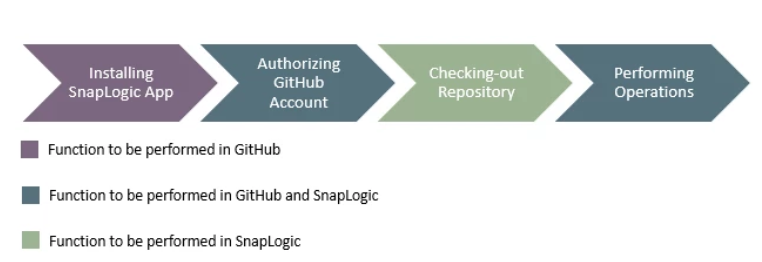
* Tag\* (required info): Automatically populated information, default value is “<version\_num> by <username> @ <date time>”. Tag information can be updated if necessary.
* Notes (required info) **(Must)**. It is always a good practice to add version notes. Note is useful for easier tracking of versions and changes. Note should contain meaningful and specific information for the version.

Please note:

* When a versioned pipeline is moved, the version history is retained.
* If a versioned pipeline is copied, the version history is not added to the new pipeline.
* During the process of creating or updating a task, only current version of a pipeline can be selected.

### GitHub versioning

SnapLogic supports GitHib integration and version control. As Snaplogic already has a good versioning of the pipelines there is a concern with tracking history of the deleted pipelines.

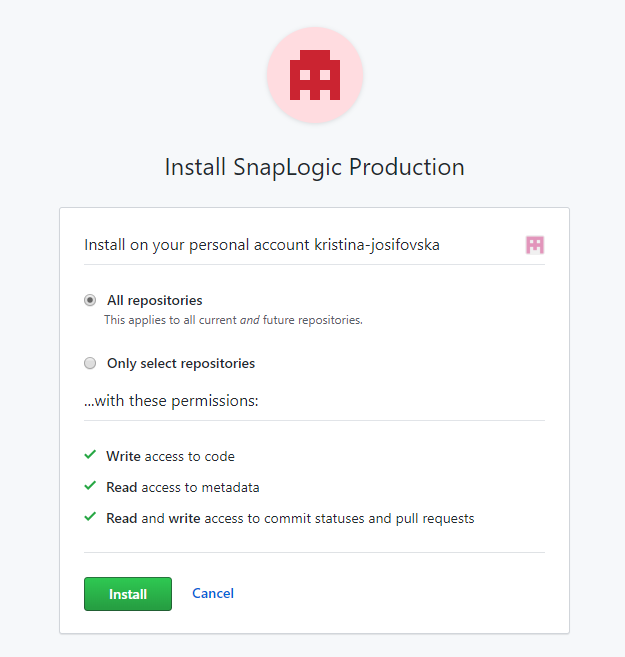


*Figure 9 – GitHub versioning flow*

Snap Packs and Snaplexes cannot be versioned in GitHub. Snap Packs and Snaplexes are listed as Unsupported (GitHub Status) assets in the Asset table.

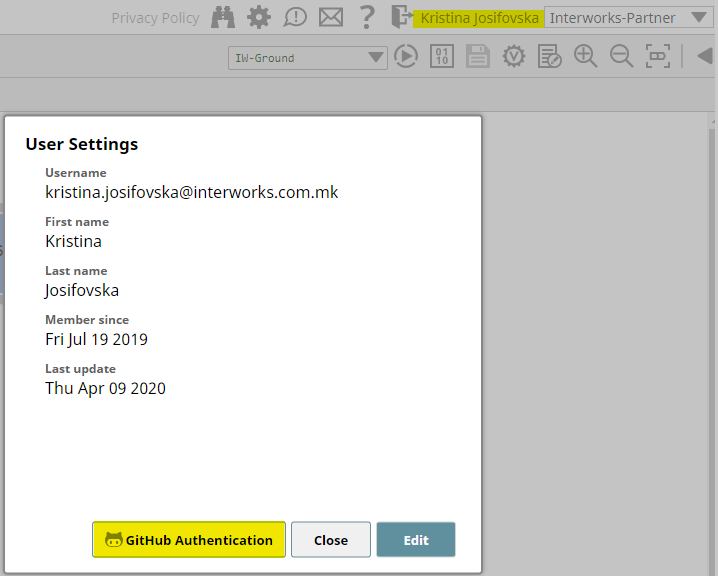
Following steps describe the process of integration SnapLogic account with GitHub repository:

1. Install SnapLogic app in GitHub



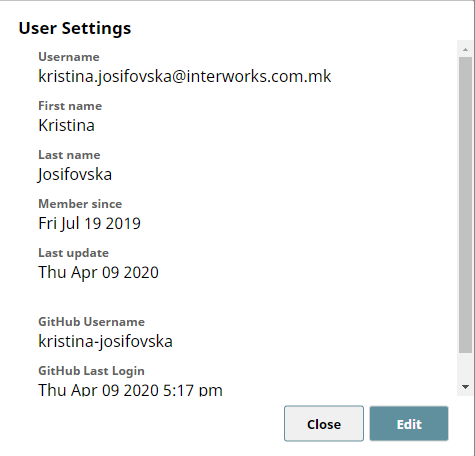
*Figure 10 – SnapLogic GitHub app*

1. Navigate to the SnapLogic profile on the top-right corner and open the User Settings dialog box then select GitHub Authentication button.



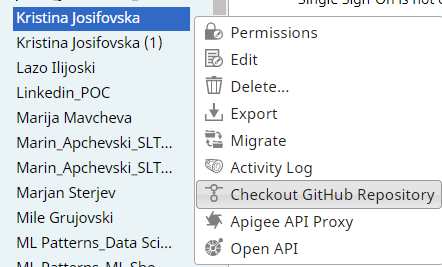
*Figure 11 – User settings dialog*

1. Login in GitHub and authorize the SnapLogic app
2. User settings should be updated with associated GitHub username

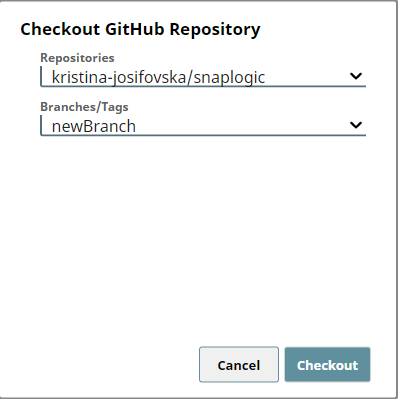


*Figure 12 – GitHub authorization in User settings*

1. From the SnapLogic Manager tab navigate the project that needs to be versioned in GitHub. Select action Checkout GitHub Repository and then in the next window there will be two drop-down boxes Repositories and Branches/Tags. The Repositories drop-down menu contains a list of the repositories in the associated GitHub org that have the SnapLogic app installed.



*Figure 13 – Checking out GitHub repository*



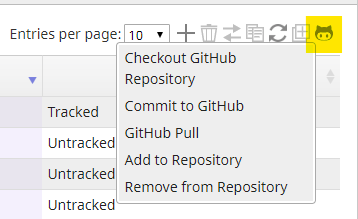
*Figure 14 – Selecting GitHub branch/tag*

1. In Manager a GitHub icon will be added to the organization’s Asset for the project. From the GitHub button actions such as checkout GitHub repository, commit to GitHub, pull, add and remove from repository are available.



*Figure 15 – GitHub tracked project*

1. Figure 15 shows versioning details where **kristina-josifovska/snaplogic/heads/newBranch** is the path of the GitHub repository, and **commit: 12f2bd7** is the Commit ID. Clicking on the commit ID navigates to the commit’s details page in GitHub. This is updated with each GitHub commit and pull operations. Additionally, the username of the GitHub account that made the last commit is displayed.



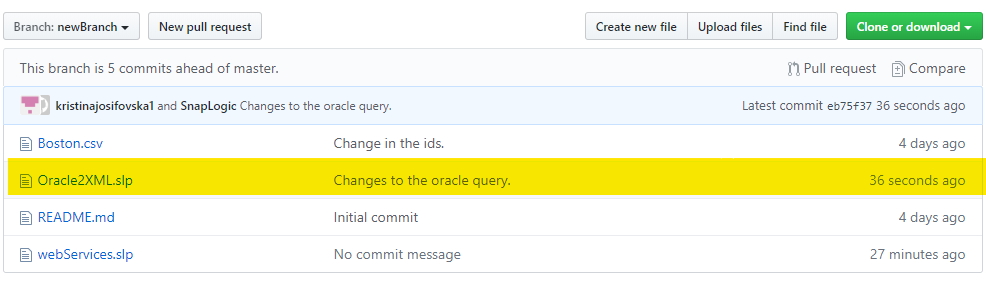
*Figure 16 – GitHub actions*

1. Pipeline changes can be committed or pulled. Additionally, a user can propose and collaborate on changes, add to the repository or remove the pipelines that are already versioned in the GitHub repository.

Following GitHub operations are available:

1. Checkout GitHub repositories. This action opens the Checkout GitHub Repository dialog box. GitHub API can also be used for this operation.
2. Commit changes in the GitHub repository. This action opens the Commit Changes dialog box.
3. Fetch updates from the GitHub repository. This action opens the Pull Messages dialog box. GitHub API can also be used for this operation.
4. Add asset to the GitHub repository. This action updates the GitHub status to Added. Next a commit is required to upload the changes in the GitHub repository.
5. Remove asset from the GitHub repository. This action changes the GitHub status to Removed. A commit operation is required to reflect the changes in the GitHub repository.

Pipeline commit operation adds the pipeline to the Asset table and the GitHub Status reads Tracked, then the new file/pipeline is displayed in the repository along with the commit message.



*Figure 17 – GitHub commit info*

## Pipeline parameters

The usage of pipeline parameters is essential during the development process in SnapLogic, it allows dynamic and clean code that can be easily maintained, adjusted and reused. Pipeline Parameters are defined in the Parameters section of the Edit Pipeline dialog (for more information on how to set up and use Pipeline Parameters, please check the official SnapLogic documentation). Once defined, parameters can be used throughout the code.

**(Must)**Best practice is to replace the hardcoded values with pipeline parameters. Parameters are available for use anywhere trough the expression language (Mapper, Pipeline Execute, JSON Generator). There is a minor difference in invocation of the pipeline parameters in some different snaps:

1. Mapper – \_parameterName
2. JSON Generator – ${\_parameterName}

During the pipeline execution, the parameters will use hold their value, until a new value arrives dynamically from a parent Pipeline or from outside call (in cases where the Pipeline is exposed as a task – API).

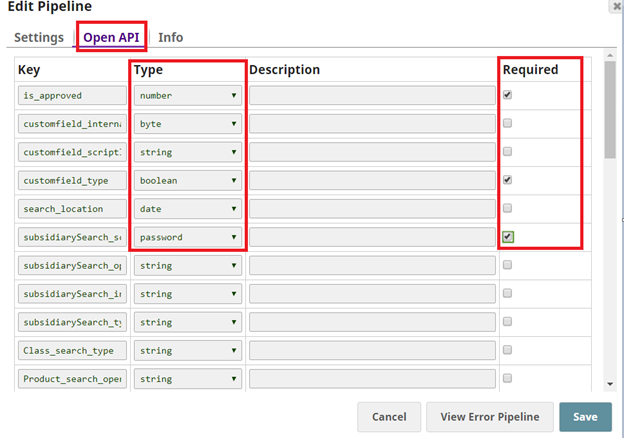
Following are some example scenarios where pipeline parameters are useful:

* Dynamically passing name of file. If pipeline use dynamic file name in multiple points, then instead of changing file name at each point in the code, define fileName pipeline parameter and use it instead. When the file name value is changed, then only value of the parameter should be changed and that will be reflected everywhere.
* Inbound/outbound email communication. Email account can be parameterized and later changed. Instead of changing the hardcoded email account throughout the code, use sendEmailTo pipeline parameter and when needed only update its value.

Good practice is to enable Capture parameter option that will allow better tracking and monitoring in runtime pipeline execution. Capture can be used on file name, Email Id, parameter that determines the flow.

After parameters have been created, in the OPEN API selection we can additionally change the parameters. Depending on the selection in the Type drop down menu, parameters can have the selections (data types): String (default), byte, number, boolean, date, password etc.

An additional Description can be added to the parameter, we can also choose whether the field is Required.



*Figure 18 – OPEN API selection*

**Important notes**

1. Use the Pipeline Execute Snap for large Pipeline parameters  
   Do not use pipeline parameters to pass excessive data, use Pipeline Execute instead. Pipeline parameters should be used to pass metadata — not data. Data can be stored in a file, and then file name can be passed as a Pipeline parameter. Similarly, when using the Pipeline Execute Snap, the data can be passed through the input view of the child Pipeline.
2. Use Expression Libraries instead of many Pipeline parameters  
   Also, when we need to set up many pipeline parameters in our code, best practice is to organize these parameters in an expression library.

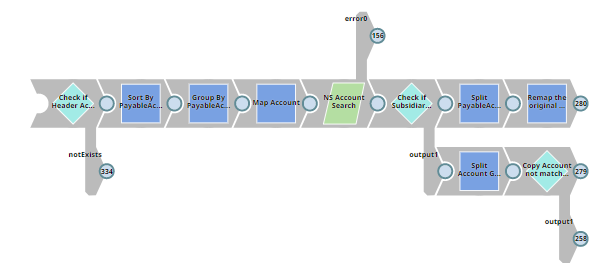
## Clean and optimized code

Keeping clean code is a very important aspect of the development process. Poorly written code is affecting the development organization, and it is hard for maintaining. During the development process in SnapLogic, clean code can be achieved by following rules:

1. Removing all unused pipelines
2. Remove all unused snaps
3. Remove all unused pipeline parameters
4. Remove all unused accounts
5. Reduce the number of snaps (whenever possible)
6. Optimizing data flow and computing operations

For performance optimization follow the advices below:

1. Reduce the number of Mapper Snaps in the pipeline, whenever possible. If possible do the same data transformations in one Mapper. This component is essential on one side but also high memory and cpu consuming on the other side. Less Mapper snaps will improve the overall pipeline performance.
2. Delete document fields in the beginning. This way less memory will be required from that point forward.
3. Filter the data before Join snap and not after. Join is one of the most expensive operations, so less data to join means faster joins.
4. Use simpler expressions over more complex ones. For example, use string split() instead of regex.
5. Use a Sort Snap immediately before a Join Snap. This will reduce the memory usage and improve performance.
6. Use Snaps instead of the expression language. For example, for data branching use Router Snap instead of a Mapper with a bunch of ternary expressions. Router snap is easier and more elegant solution.
7. Split large input data into multiple smaller chunks. This can be done with the GroupByN Snap, where preconfigured number of input documents are grouped into an output document, next with Pipeline Execute snap smaller groups can be processed one by one or in parallel. This could improve the overall performance.
8. Sending SOAP/REST Requests. Before making a SOAP or REST call, it is advised to group the call in order to avoid repeating the same calls. Example scenario: On Input we have 100K records and each record needs an account. We can use 10 existing accounts. Each record we process based on a specific ID needs to be assigned an account. If we have records that do not contain an account ID we can route them to avoid empty requests. To avoid making 100k requests, we group them by account ID and make no more than 10 requests. After Accounts are obtained from SOAP/REST, the records will split. Please check the solution below:

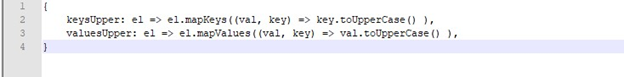


*Figure 19 – Correct solution for account splitting*

## Expression libraries

Expression Libraries are files made of one or more expressions (functions) that can be imported and used into a Pipeline. They can be used whenever expression language can be used (ex. Mapper).

Figure 20 shows an example of an expression library named dim.expr that contains two expressions (methods) named keysUpper and valuesUpper. First method keysUpper is taking an object as a parameter and it’s converting its keys into all uppercase letters. Second method valuesUpper is taking an object as a parameter and it’s converting its values into all uppercase letters.



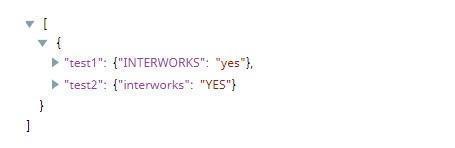
*Figure 20 – Example usage of function from expression library*

We recommend using these expressions for common functions used in multiple places. Expressions from the expression library are invoked with the use of global variable lib.

Syntax: lib.Name of the File (expression library). Name of the expression (method)  
Example: lib.dim.keysUpper



*Figure 21 – Usage of expressions for expression library*

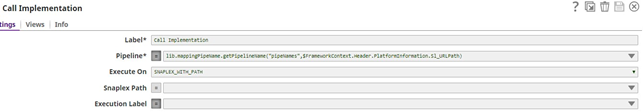


*Figure 22 – Result from execution of expression from expression library*

The expressions from expression libraries can be used as configuration files. Figure 23 shows example of configuration file with expression library.



*Figure 23 – Screenshot of Expression Library named “mappingPipeName”*



*Figure 24 – Example of usage of “getPipelineName” from configuration file*

For more detailed info go to official SnapLogic documentation.

## Error handling

Having a good Error Handling mechanism is essential in the software development process, because it prevents data loss and application failure. In this section are explained the error handling options offered by SnapLogic and how should be used.

### General error handling options

Following are the error handling options that can be used in SnapLogic:

* **Stop pipeline execution**

When error occurs and error pipeline is not implemented, default behavior in all snaps in SnapLogic is “stop pipeline execution” that will interrupt the pipeline execution.

* **Discard error data and continue**

Another option in the error dropdown menu is “discard error data and continue”, simply ignoring the error. With this option selected pipeline is not stopped, but this behavior leads to data loss and we are using it on a very rare occasions for testing purposes or unless the error thrown is trivial and should not stop the pipeline execution. Use with caution.

* **Route error data to error view**

Third option is “Route error data to error view”. This option can be attached on snap level. The pipeline will complete the execution with compromised records sent to the error view together with the detailed description of the error. On Dashboard the pipeline will be displayed as success. **(Must)** This option should be used in case when we need to predefine the error when sending it to the user.

* **Route error data to error pipeline**

This approach requires error pipeline to be implemented and if you go to the view tab of some snap that is used in the pipeline you can choose “Error data to error pipeline”.

* **Error pipeline (Must)**

Besides the options above, Error Pipeline can be used for handling errors. Please note that Error Pipeline handles the errors on pipeline level, not on an individual snap. Error pipeline is **our** **recommended error handling option**. Also, important notification that you will receive when applying error pipeline is that the error pipeline will change the behavior of the snaps that are used in that particular pipeline.

When error occur, error data is automatically passed to the Error Pipeline for further handling. Next Error Pipeline accepts the error object and handles it according to the flow.

Error pipeline can be reused for more projects, contributing to code modularity. **(Should)** It is recommended to have one Error handling mechanism, located in a shared project and ideally used in all projects.

### 5.5.2 Error message format

Using our experience and following the growth of the SnapLogic as a tool we will give you our best practice general template for error handling mechanism. This template can wary depending of the specific client needs but can be used as a starting point.

While creating this process we recommend using the following fields. You can include or exclude some fields depending of the client needs.

As we previously mentioned, this is fully customable and you can use weather to write to the DB, File, create some incident ticket, send email notification… or a combination of all options.

|  |  |  |
| --- | --- | --- |
| Field description | Field name | Structure |
| Short name of the Application that the flow belongs to.  Exmp: ODR (Order), ITM (Item), INV (Inventory) | ApplicationEntityId | $ErrorReport.Header.EventInformation.ApplicationEntityId |
| Flow direction of the event when compared with the system integrated:  Salesforce sends message to SL: SalesforceOutbound  Billing system receives message from SL: BillingInbound | ComponentMessageDirection | $ErrorReport.Header.EventInformation.ComponentMessageDirection |
| Name of the system that the message belongs to:  Exmp: Saleforce, Billing, Netsuite etc. | ApplicationName | $ErrorReport.Header.EventInformation.ApplicationName |
| Unique system id that points to the transaction that created the event in SnapLogic:  OrderId: 123-AB-34C, ItemId: 99889\_uusdf\_123 | ApplicationTransactionId | $ErrorReport.Header.EventInformation.ApplicationTransactionId |
| Unique id generated in SnapLogic that will correlate single event/transaction across multiple systems.  Build pattern:  Math.randomUUID() + "-" + ApplicationEntityId + "-" + ApplicationName | CorrelationId | $ErrorReport.Header.EventInformation.CorrelationId |
| Operation that describes the Flow invoked:  Exmp: CreateOrder, DeleteItem, UpdateInventory | EntityOperationName | $ErrorReport.Header.EventInformation.EntityOperationName |
| Datetime when event/job occurred. | EventDateTime | $ErrorReport.Header.EventDateTime |
| Used to make distinction of the level of information stored in the AuditLog table.  1. Value is set to debug: Complete payload and parameters will be stored in the AuditTable.  2. Value is set to audit: payload and parameters will NOT be stored in the AuditTable.  3. Value is set to error: no logic with this type. Only info will be stored in the ExceptionTable. | EventLogType | $ErrorReport.Header.EventInformation.EventLogType |
| Used to save the execution state of the pipeline in the AuditLog table.  Values: Started, Completed, Error | EventLogStatus | $ErrorReport.Header.EventInformation.EventLogStatus |
| Pipeline name | SL\_PipeLineName | $ErrorReport.Header.PlatformInformation.SL\_PipeLineName |
| Unique Id of the execution of the pipeline.  Automatically generated in SnapLogic. | SL\_PipeLineRunID | $ErrorReport.Header.PlatformInformation.SL\_PipeLineRunID |
| Unique Id of the execution of the parent pipeline (if any).  Automatically generated in SnapLogic. | SL\_PipeLineParentRunID | $ErrorReport.Header.PlatformInformation.SL\_PipeLineParentRunID |
| Unique Id of the pipeline.  Automatically generated in SnapLogic. | SL\_PipeLineInstanceID | $ErrorReport.Header.PlatformInformation.SL\_PipeLineInstanceID |
| Full path of the project to which the pipeline belongs to in SnapLogic. | SL\_ProjectPath | $ErrorReport.Header.PlatformInformation.SL\_ProjectPath |
| The machine on which the Snaplex that runs the pipelines resides. | SL\_Host | $ErrorReport.Header.PlatformInformation.SL\_Host |
| Path to the Snaplex that runs the pipelines. | SL\_PlexPath | $ErrorReport.Header.PlatformInformation.SL\_PlexPath |
| The user that runs the pipeline. | SL\_User | $ErrorReport.Header.PlatformInformation.SL\_User |
|  | SL\_URLPath | $ErrorReport.Header.PlatformInformation.SL\_URLPath |
| OriginalPayload (optional, if there is no log start to keep this value add OriginalPayload) | OriginalPayload | $ErrorReport.Body.SL\_OriginalPayload |
| Error code returned | ExcepCode | $ErrorReport.ExcepCode |
| The error reason element returned from SnapLogic. | ExcepShortDescription | $ErrorReport.ExcepShortDescription |
| The error StackTrace element returned from SnapLogic. | ExcepStackTrace | $ErrorReport.ExcepStackTrace |
| “Failed” | ExcepMessage | $ErrorReport.ExcepMessage |
| The error resolution element returned from SnapLogic. | ExcepResolution | $ErrorReport.ExcepResolution |
| The label of the failed snap | ExcpFailedLabel | $ErrorReport. ExcpFailedLabel |
| If there is a retry functionality (add this field) | RetryCount | $ErrorReport.RetryInformation.RetryCount |
| If there is a retry functionality (add this field as a flag that can help you building the retry logic) | IsEventRetried | $ErrorReport.RetryInformation.IsEventRetried |

Sample of error file:

{

"ErrorReport": {

"Header": {

"EventDateTime": "2020-01-16T01:25:14.865Z",

"EventInformation": {

"ApplicationEntityId": "ODR",

"ApplicationName": "NetSuite",

"EntityOperationName": "CreateOrder",

"ComponentMessageDirection": "NetsuiteOutbound",

"ApplicationTransactionId": "123-AB-34C",

"CorrelationId": "7e37894f-19d8-4613-b8ed-54cc89888b1c",

"EventLogStatus": "Failed",

"EventLogType": "audit"

},

"PlatformInformation": {

"SL\_Host": "pa23SL-ijccux-0607bb4b029914f9c.pia.snaplogic.com",

"SL\_ProjectPath": "/Test/Test\_DEV/Util",

"SL\_PipelineName": "01\_p1\_template\_g2",

"SL\_PipelineInstanceID": "7e37894f-19d8-4613-b8ed-54cc89888b1c",

"SL\_PipelineRunID": "5e0af9f26baeec19658d84f2\_1991e2dc…", "SL\_PipelineParentRunID":”5e0af9f26baeec19658d84f2\_1991e2d",

"SL\_PlexPath": "/Test/shared/Cloud",

"SL\_User": "gmirchevski@snaplogic.com"

}

},

"Body": {

"OriginalPayload": "Some text"

}

},

"ExcpCode": "String",

"ExcpShortDescription": "String",

"ExcpReason": "String",

"ExcpResolution": "String",

"ExcpStackTrace": "String",

"ExcpFailedLabel": "String",

"ExcpFailedPipeline": "String",

"RetryInformation": {

"RetryCount": "2",

"IsEventRetried": "false"

},

"CustomFields": {

"Parameter": {

"parameterName": "parameterValue",

"parameterName2": "parameterValue2"

}

}

}

}

SQL Structures location:

Pattern error handling template location:

## Logging

Sometimes there is a requirement for logging information for pipeline execution such as start and stop of the process, error message, etc. Our advice is to use a file or database for this purpose and store it locally or on a cloud. When logging the pipeline information in a file, good practice is to put a timestamp in the name of the file in order to know the exact time of creation. Recommendation is to log only the parent level of the process, logging the sub-pipelines is not necessary.

The structure of the audit tables is overlapping with the structure of the exception table. For that manner please refer the above table.

Sample of log start file:

{

"AuditReport": {

"Header": {

"EventDateTime": "2020-01-16T01:25:14.865Z",

"EventInformation": {

"ApplicationEntityId": "ODR",

"ApplicationName": "NetSuite",

"EntityOperationName": "CreateOrder",

"ComponentMessageDirection": "NetsuiteOutbound",

"ApplicationTransactionId": "123-AB-34C",

"CorrelationId": "7e37894f-19d8-4613-b8ed-54cc89888b1c",

"EventLogStatus": "Started",

"EventLogType": "audit"

},

"PlatformInformation": {

"SL\_Host": "pa23SL-ijccux-0607bb4b029914f9c.pia.snaplogic.com",

"SL\_ProjectPath": "/Test/Test\_DEV/Util",

"SL\_PipelineName": "01\_p1\_template\_g2",

"SL\_PipelineInstanceID": "7e37894f-19d8-4613-b8ed-54cc89888b1c",

"SL\_PipelineRunID": "5e0af9f26baeec19658d84f2\_1991e2dc…",

"SL\_PipelineParentRunID": "5e0af9f26baeec19658d84f2… ",

"SL\_PlexPath": "/Test/shared/Cloud",

"SL\_User": "gmirchevski@snaplogic.com"

}

},

"Body": {

"OriginalPayload": "Some text",

"OriginalParameters": {

"Parameter": {

"parameterName": "parameterValue",

"parameterName2": "parameterValue2"

}

}

},

"CustomFields": {

"Parameter": {

"parameterName": "parameterValue",

"parameterName2": "parameterValue2"

}

}

}

}

You will notice that Header object of the JSON is exactly the same as it is in the Error handling structure. Following fields are excluded from the Audit logging structure since they are strictly related to the exception part.

* ExcepCode
* ExcepShortDescription
* ExcepStackTrace
* ExcepMessage
* ExcepResolution
* ExcpFailedLabel
* RetryCount
* IsEventRetried

You will notice that in the Body JSON object of the sample above there is a field Original Payload and also additional object inside “Body” OriginalParameters.Parameter that should carry the input parameters of the process if that is required. Same as the exceptions, the structure of the audit (file, DB table…) can wary depending on the client needs but you should keep using the above naming of the fields. If there is a requirement to add some new fields please follow the naming convention similar to the above JSON samples.

In order to get specific information for the Pipeline, such as the pipeline runtime id, use Pipeline Functions from the Expression Language. Please check the official SnapLogic [documentation](https://docs-snaplogic.atlassian.net/wiki/spaces/SD/pages/1439352/Pipeline+Functions+and+Properties) for all available Pipeline Functions and Properties.

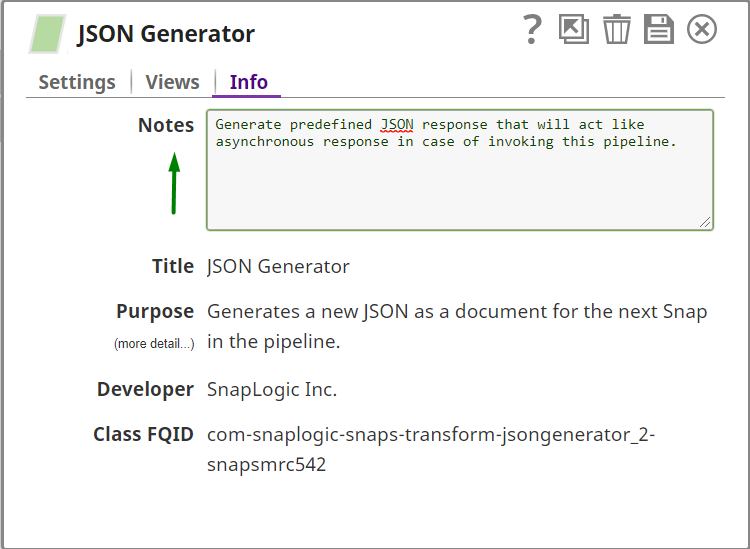
SQL Structures location:

Pattern error handling template location:

## Documenting the code

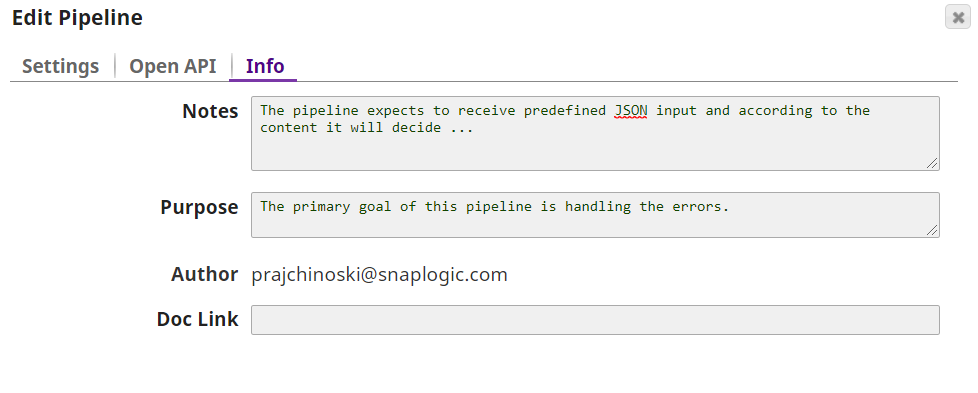
A good development practice besides the naming of the snaps, pipelines, accounts, tasks… is applying comments on the code while developing. In SnapLogic it can be done on the following ways.

**(Should)** If the label of the snap does not give us the whole explanation of what that particular snap is doing then we should add more details in the Info > Notes of the snap.

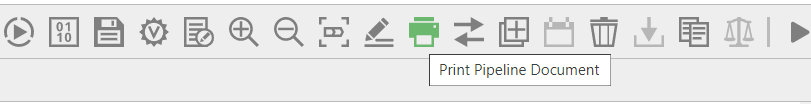


**(Should)** Similar to the snap info tab, we have Info tab on the pipeline level. Add a pipeline overview that will descriptive enough to cover the core functionality of the pipeline.

Edit Pipeline > Info



**(Can)** Once the development is done, along with the documentation that the developer should provide, we can add the autogenerated detailed documentation directly from the pipeline in pdf format. If there are notes ..



# Code Migration strategy

Code migration is one of the key phases of the SDLC. Defining a good migration strategy will definitely pay off in the long run, primarily because it saves time and resources. There are different ways for migrating the code and our recommended approach does not necessarily mean it will always fit the client needs, mainly because it depends of few factors like: number of organization available, projects complexity and size, usage of GitHub.

This being said, we will explain the following migration approaches:

* Manual moving of assets, using export/import options
* Migration option
* GitHub approach
* Fully automated CI/CD

# github code maintenance using snaplogic

## Environments

The document assumes existence of the following environments:

* Development (DEV)
* Test (TEST)
* Staging (STAGE)
* Production (PROD)

Note: In this case only one environment is used. Project spaces are created that should act like separate environments.

## General notes and cautions

### General GitHub notes

Each SnapLogic project will have its own code repository.

The code management workflow introduces two long-lived branches:

* develop
* master

The branches used in the code management have the following meaning:

* Master branch: There is development in the master branch. The code that is merged into this branch is considered as already tested and production ready. The code can be merged into the master from the develop branch. **DO NOT** commit code on this branch.
* Develop branch: This is the main development branch that collects the features developed by all developers in the team.

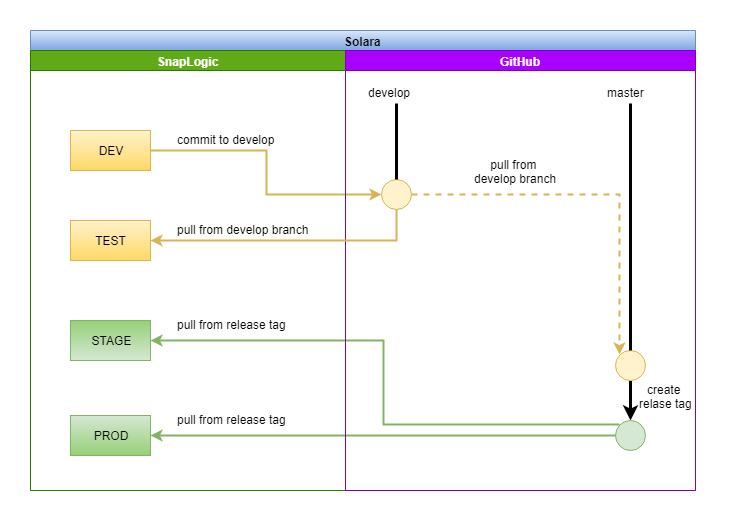
### General Snaplogic notes

This document assumes that GitHub code versioning system will be used only from the Manager tab of the SnapLogic tool.

**Important notes**:

* Repositories **must** be created before using them with SnapLogic.
* SnapLogic projects **must** be created manually in each environment (DEV, TEST, STAGE, PROD)
* Only pipelines and files should be versioned using this approach.
  + Committing Task in the repository can produce unlikely scenarios.
    - Recommendation: Create the tasks manually in the upper environments.
      * If there are a lot of task and you want to keep them in GitHub you can commit tasks as well but once they are deployed on an upper environment **you must check the task settings in the manager tab on each upper environment.**
  + DO NOT commit accounts in the GitHub repository.
    - Recommendation: Create accounts manually in the upper environments.

## Workflow diagram

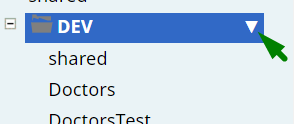
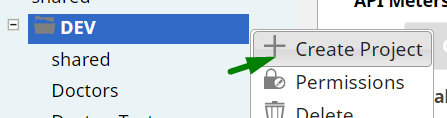
The following diagram describes the connection and migration path between SnapLogic and GitHub.

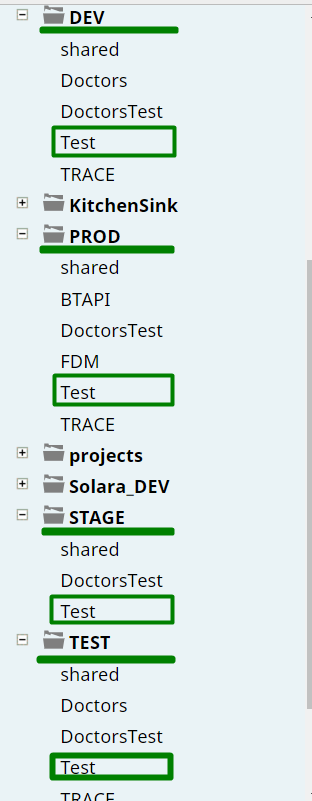
## Step by step migration guide

### Create Snaplogic Projects

Before starting any development in SnapLogic you must make sure that the pipeline that will be created are placed in appropriate project. Each project space has its own default shared project that is automatically created while creating the project spaces. Use this shared project to store pipelines, accounts, files that will be shared among all other projects in that appropriate project space.

Project in SnapLogic can be created in the Manager tab.

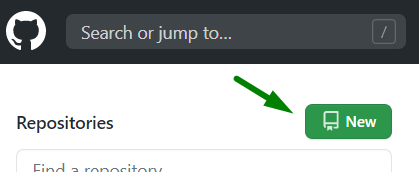
1. Choose the project space
   1. 
2. Choose Create Project
   1. 
3. Enter the name of the project and hit Save.
4. **Important: Create the project in all four environments. Use the same project name in each environment.** See the bellow example. The project “Test” is created in all four environment with the same name.



### Create GitHub repository

Once you have the project created in SnapLogic then you need to create Repository in your GitHub environment.

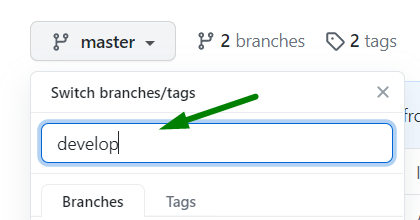
For simplicity, we will use GitHub page where you can create the repository really fast.

1. Click on the “New” button
   1. 
2. Enter the name, description, privacy... Click on Create repository button.
   1. Name the repository same or similar as your SnapLogic project.

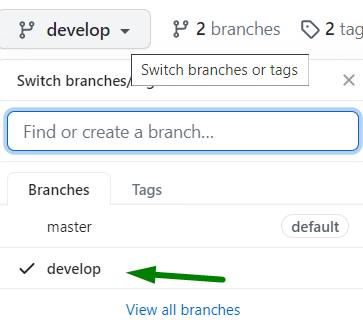
### Create develop branch

Once the repository is created, in the branch part, you will notice only one branch ‘master’. We need to keep that branch and create additional ‘develop’ branch.

1. In the branches/tags search section type ‘develop’ and hit enter on the keyboard.

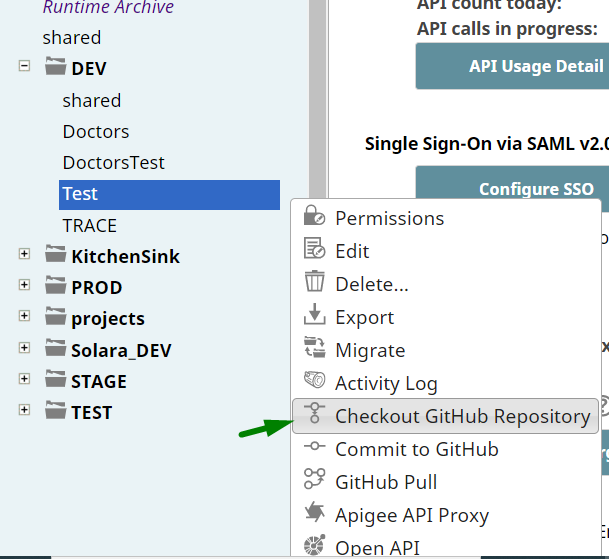


1. If the branch is created you should be able to see that branch in the same search window.

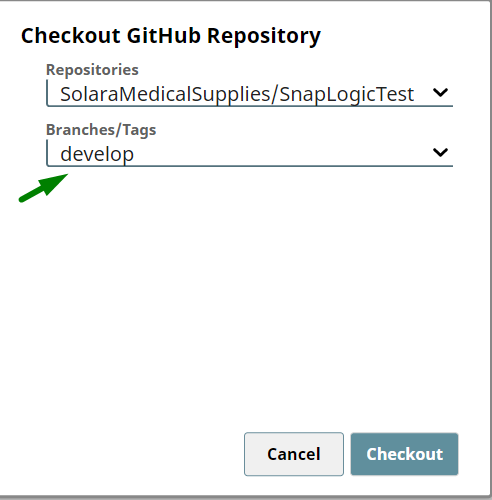


### Checkout DEV and TEST Snaplogic projects to develop branch

1. Choose ‘Checkout GitHub Repository’ option for the appropriate project



1. Use develop branch from the appropriate repository

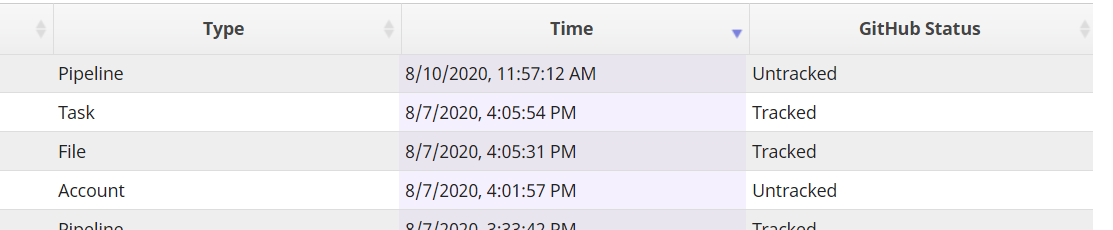


Do the same for the project in TEST environment/project space.

### Commit assets on develop branch

Once you are connected to the develop branch you can start committing assets from the DEV project. Do not commit assets from another project except from the development prject.

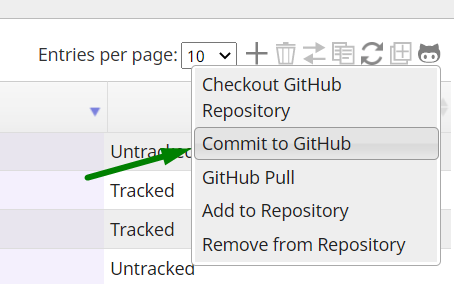
To check the GitHub status, you can open the Manager -> Navigate to the project -> Check the GitHub Status column



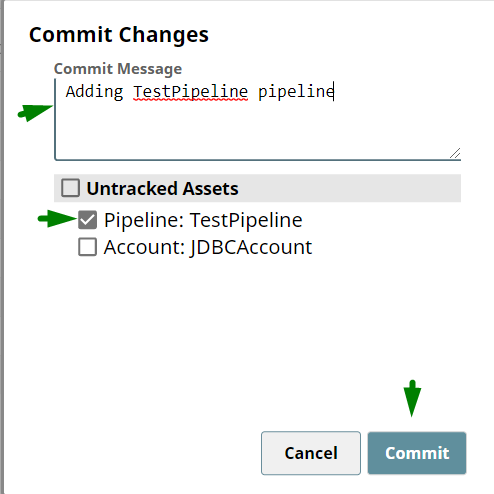
The items that are not committed to the repository are marked as untracked. The first time you should expect all of the assets to be untracked.

To commit asset or more assets on the repository you should:

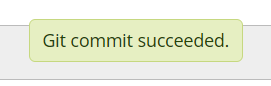
1. Choose Commit to GitHub option



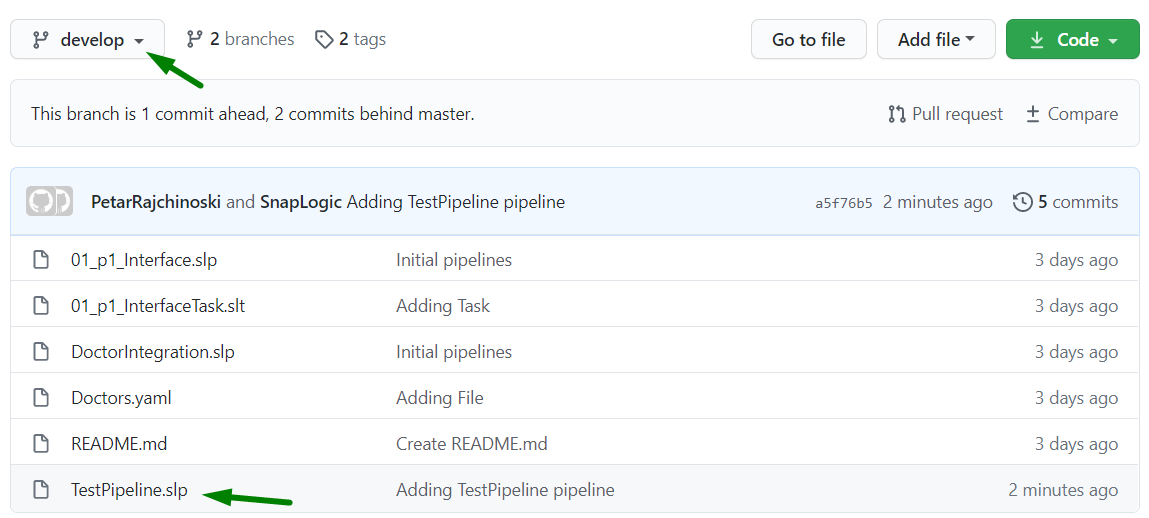
1. Select the asset/assets you want to commit, add commit message and hit the Commit button



1. Wait for the confirmation message at the same windows in SnapLogic



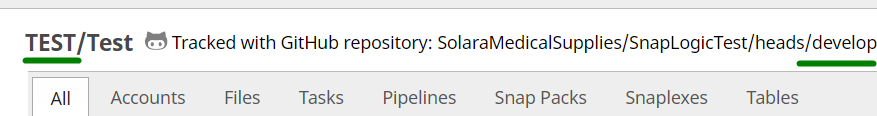
1. Check committed assets on GitHub



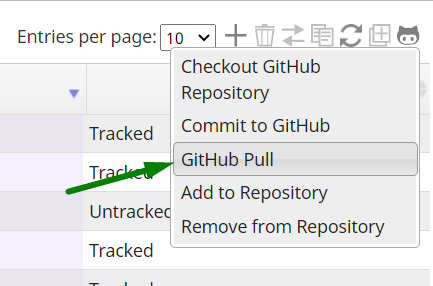
### Pull from develop branch

To pull current code from the develop branch to the test project in TEST SnapLogic environment do the following steps.

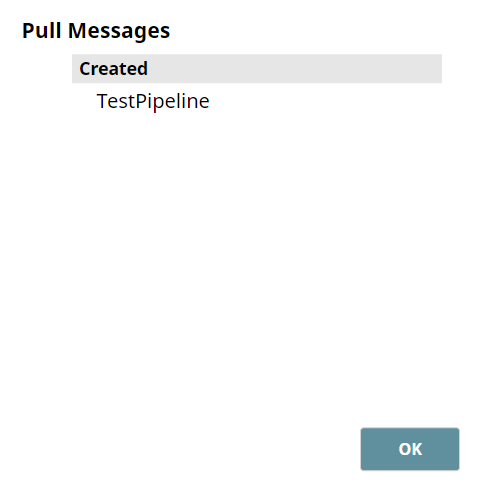
1. Make sure you are at the develop branch



1. Checking accounts status
   1. Make sure you have the Accounts manually created in your project in the TEST, STAGE, PROD environments
2. Choose Pull from GitHub and wait while the pulling is done







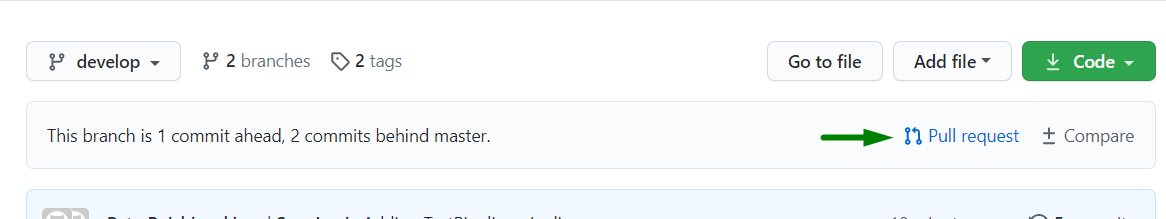
1. Reload (refresh page) your SnapLogic project to see the pulled assets.
2. Checking Tasks status:
   1. If you **are committing/pulling** tasks in the project
      1. After the pulling from the develop branch to your project in the TEST environment **make sure the settings of the task are correct**
   2. If you are **NOT committing/pulling** tasks, you are creating them manually in your TEST environment project

Make sure to create the task/tasks

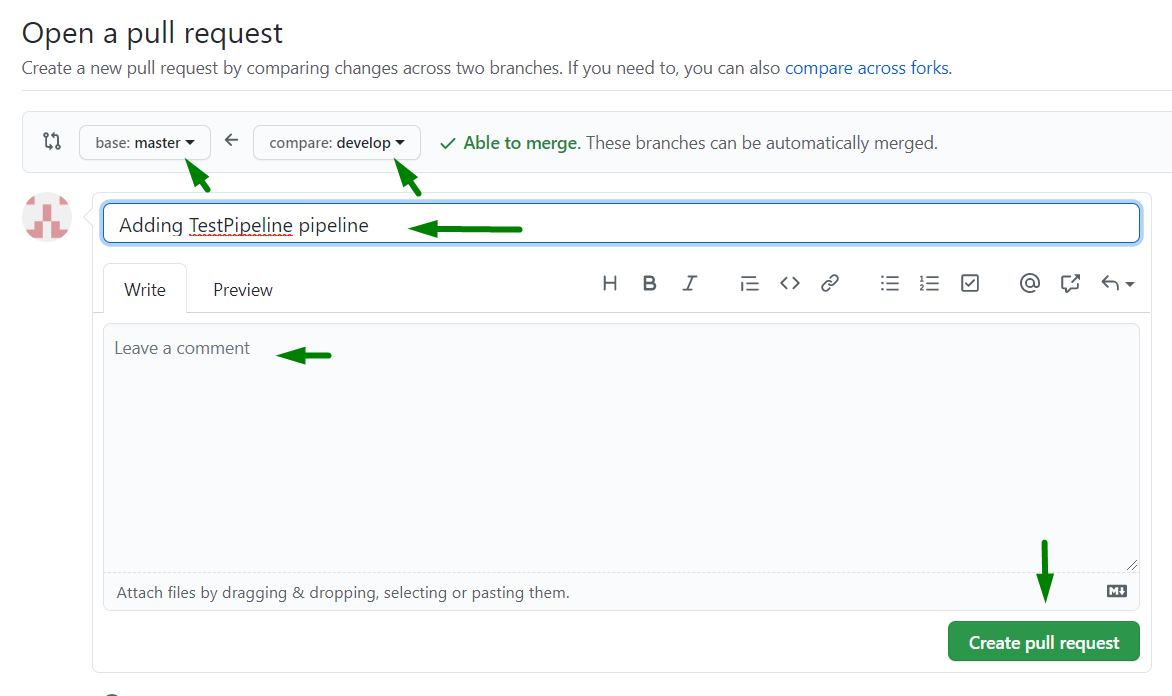
### Merge master branch

Once the code is tested and production ready we need to move it to the master branch. For that purpose, one of the easiest ways of doing this is:

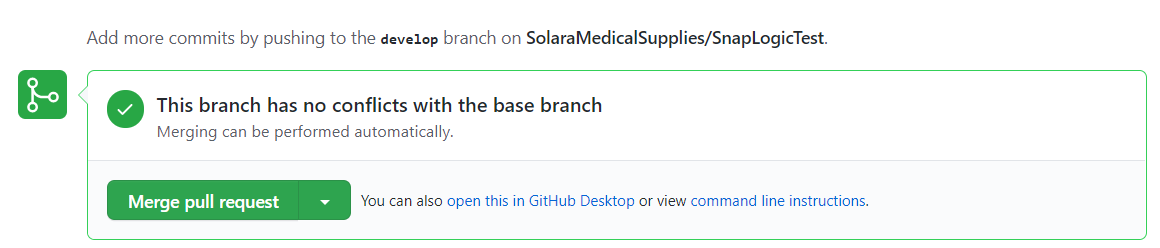
1. Go to the GitHub repository
2. Click on the Pull request link



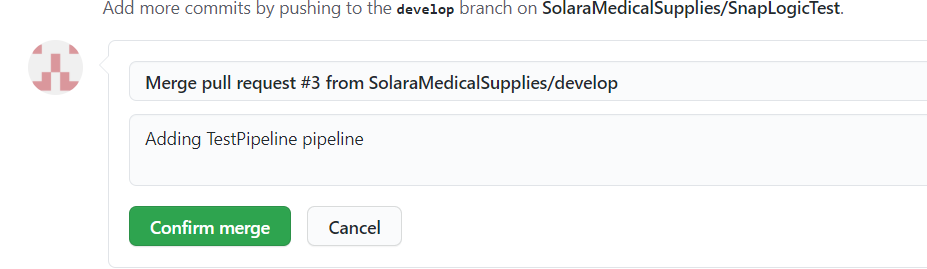
1. Set up the pull request and click on the Create Pull Request button. It should be from develop to master. See the picture below



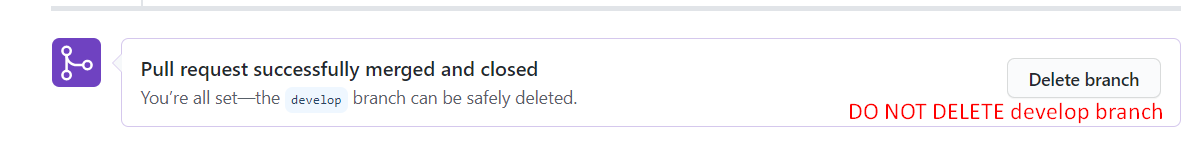
1. Click on the Merge pull request



1. Confirms Merge



**DO NOT DELETE** develop branch

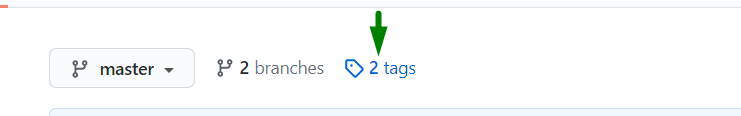


1. Check the master branch to see if everything is fine

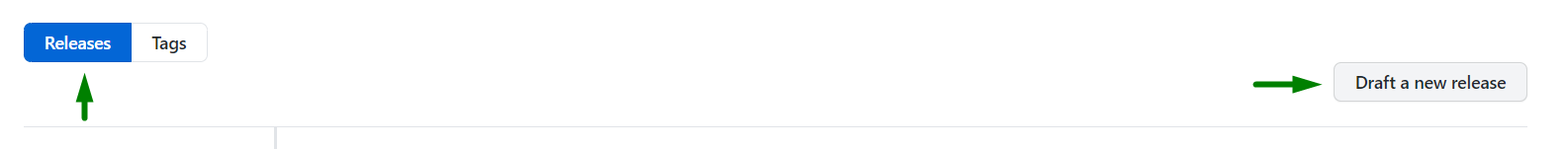
### Create release tag

Once the code is pulled on the master branch, we need to create release tag.

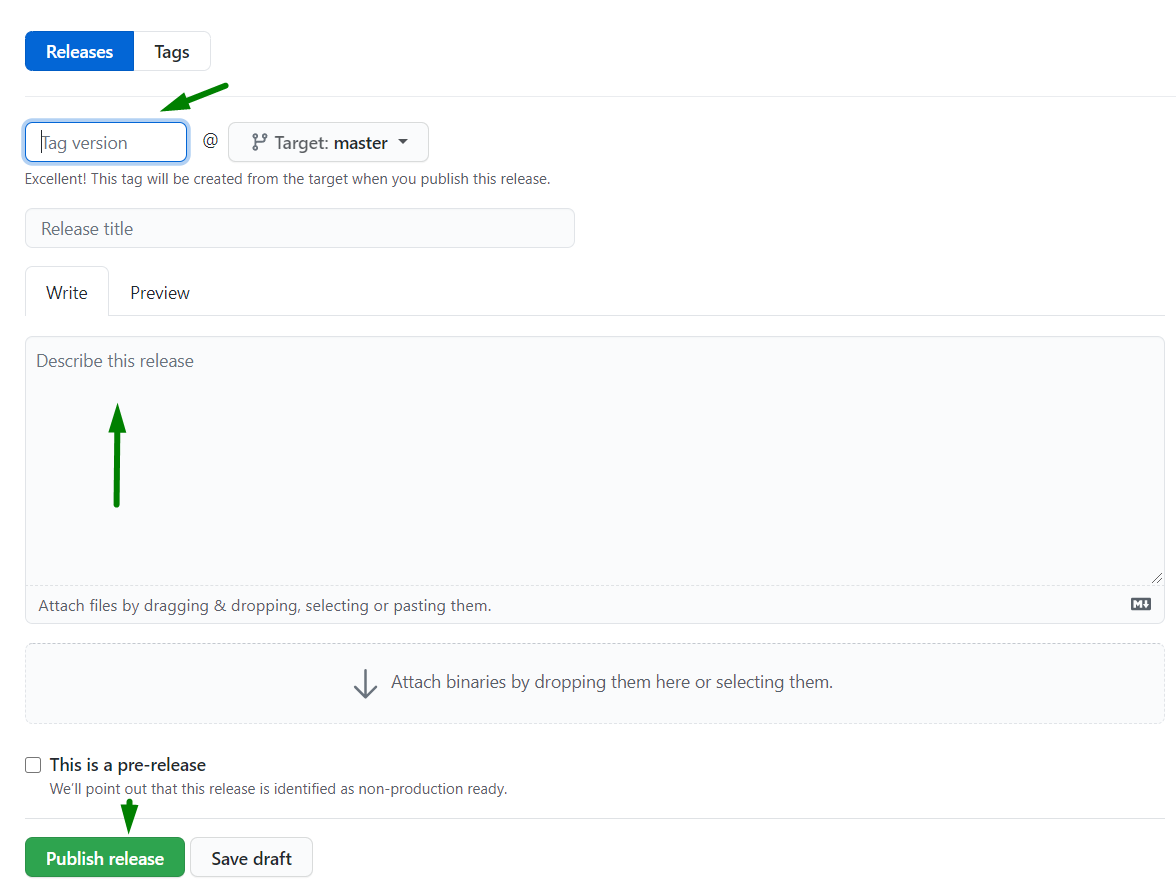
1. Open tags section in GitHub



1. Click on the Draft new release



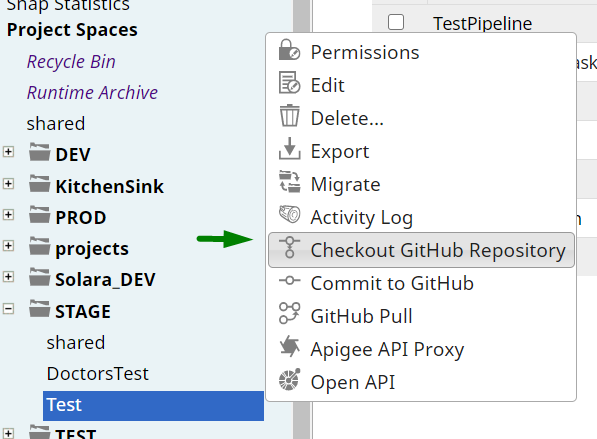
1. Set up release details, and click on Publish release



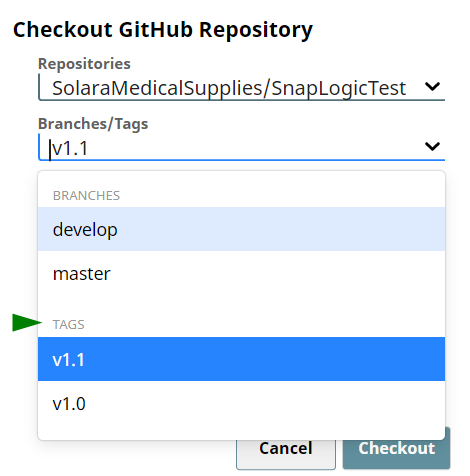
1. Check the release in the releases window

### Checkout STAGE and PROD on release tag

1. Checking accounts status
   1. Make sure you have the Accounts manually created in your project in the STAGE, PROD environments
2. Checkout GitHub repository



1. Select the appropriate tag



1. Check/verify the code



1. Checking Tasks status:
   1. If you **are committing/pulling** tasks in the project
      1. After the pulling from the develop branch to your project in the TEST environment **make sure the settings of the task are correct**
   2. If you are **NOT committing/pulling** tasks, you are creating them manually in your TEST environment project
      1. Make sure to create the task/tasks

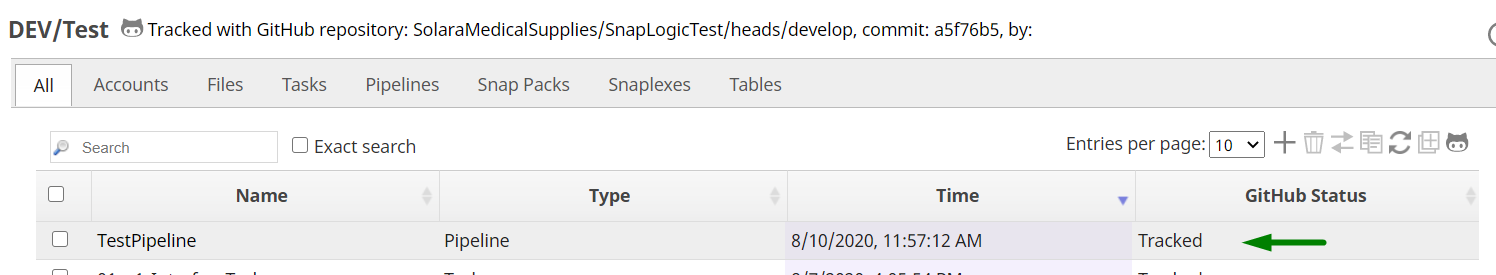
Once Stage code is approved you can pull the code in PROD using the same steps above.

### Delete/Remove assets from SnapLogic and GitHub

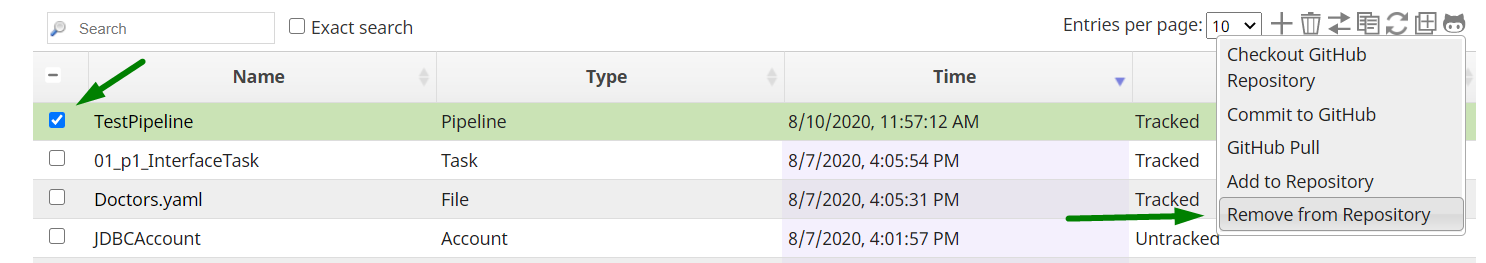
If you have some asset (pipeline/file/task/account) that you want to remove it from the SnapLogic project but also to remove it from the appropriate GitHub project follow the next steps.

**You are safe to delete assets with status “Untracked”** otherwise you might create a conflict.

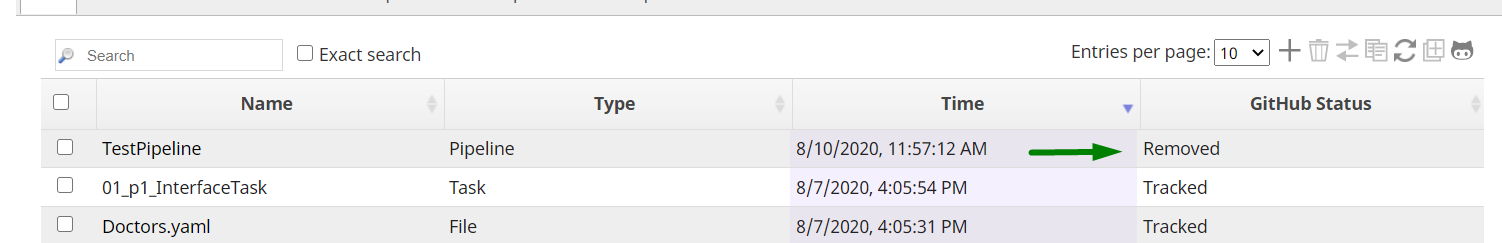
Note: In the following steps we will delete the “TestPipeline” from SnapLogic but we want it to be removed from the GitHub repository as well.



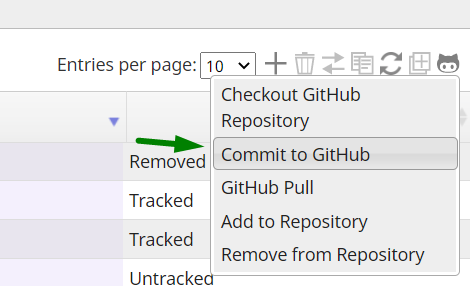
1. Select the asset and click on the “Remove from Repository”

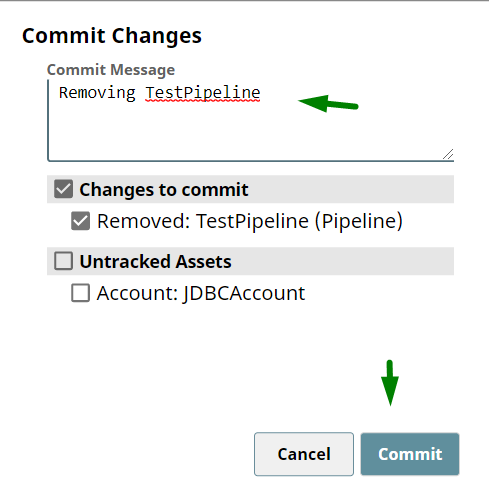


1. Wait until you see status “Removed”

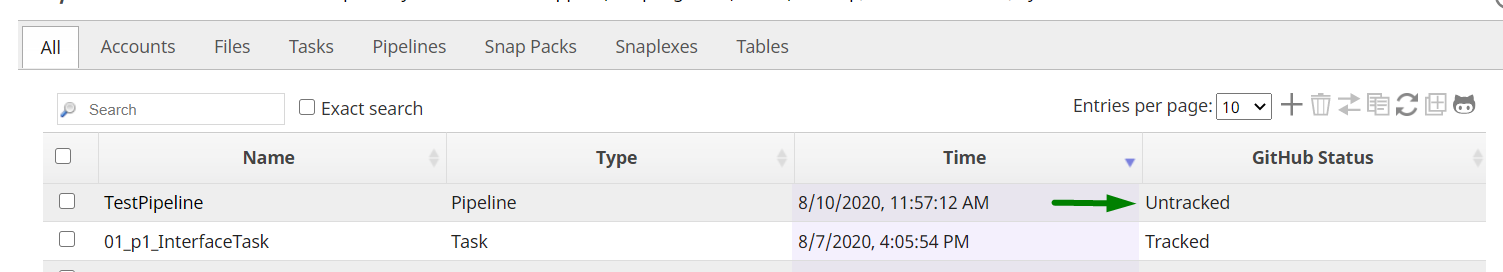


1. Commit the removal to GitHub

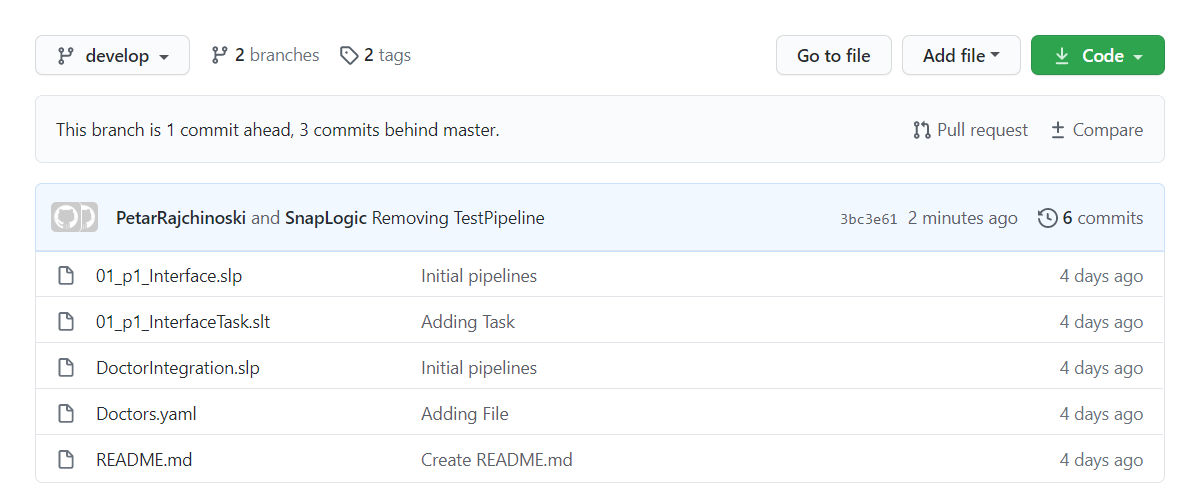




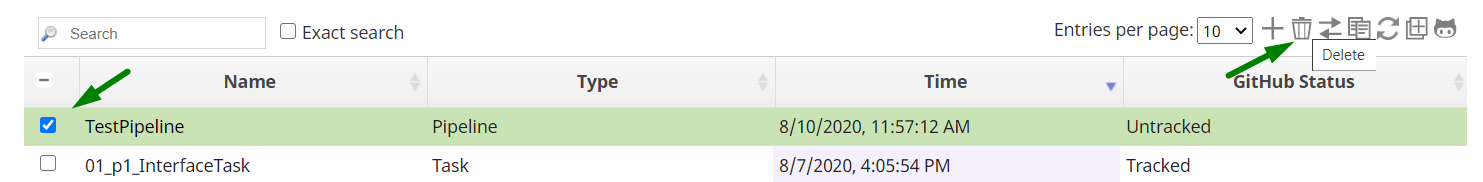
1. Check GitHub status (after the commit it should be “Untracked”)



1. Check GitHub Repository
   1. We can see that “TestPipeline” no longer exists in the develop branch



1. Delete the asset from SnapLogic
   1. Now you are safe to delete the asset from SnapLogic if it is needed.



# Adminstration

## Managing users and groups

By default, each SnapLogic organization has two groups of users:

* admins - have full access to all projects in the organization
* members - have access only to projects that they create.

Groups allow you to manage project permissions on a higher level than specific users, and this is really handy when you have two or more different teams or somehow separated developers that are working on the same organization.

**(Can)** If you are admin you can create separate groups and add all users to the appropriate group. That way by only giving access to the group you will automatically add access to each member of that group. Also, it is easy to add and remove group members so and change the permissions to the group.

More about groups and users on the [official SnapLogic documentation](https://docs-snaplogic.atlassian.net/wiki/spaces/SD/pages/1439127/Administration+Configuration).Not sorted section: TBD TBD

# Microservices

Pipelines are services or sets of actions that process data in sequence. They can be seen as microservices since they are performing specific functionalities and they can be exposed as tasks (APIs) with URL and authorization details and then invoked from outside systems.

Each Pipeline can be exposed as a specific task:

1. Triggered – Executed by accessing a URL
2. Scheduled – Executed on a specific time interval
3. Ultra – High-availability and low-latency request/response API

When the Pipeline is exposed as a Triggered Task its execution response must be in line with the client's requirements, and this must be set up specifically in the Pipeline. In order to achieve this, we should use a Mapper Snap, followed by a JSON Formatter Snap at the end of the Pipeline. In the Mapper Snap we should configure the appropriate response of the Pipeline, which must describe the final action done by the Pipeline. In the response we should also set up a status code. The status codes that are sent by the Pipelines, exposed as microservices, should follow the standard HTTP response status codes. In order for this status code to be sent correctly, we should use a JSON Formatter Snap after the Mapper Snap, where we can pass the status code as a final output of the Pipeline.

On the official SnapLogic [documentation](https://docs-snaplogic.atlassian.net/wiki/spaces/SD/pages/1438189/How+to+Run+a+Pipeline+from+a+URL+Triggered+Task) you can find out more about the handling of the execution response and the status code when your Pipeline is exposed as a triggered task.

# Child pipelines

In case when same combination of snaps is used multiple times, it is recommended to group the snaps into a separate pipeline and invoke it when needed. This pipeline is called a child pipeline, sub pipeline, nested pipeline or subprocess, because it’s invoked from the main (parent) pipeline. Main benefit of child pipelines is code reusability. We advise to split the pipelines into sub-pipelines when:

* General pipeline (main) can be used for multiple processes, i.e. to invoke different child pipelines depending on the job that is processing.
* Sending parameters dynamically in sub-pipelines for multiple uses.
* Splitting the integration for performance optimization, etc.

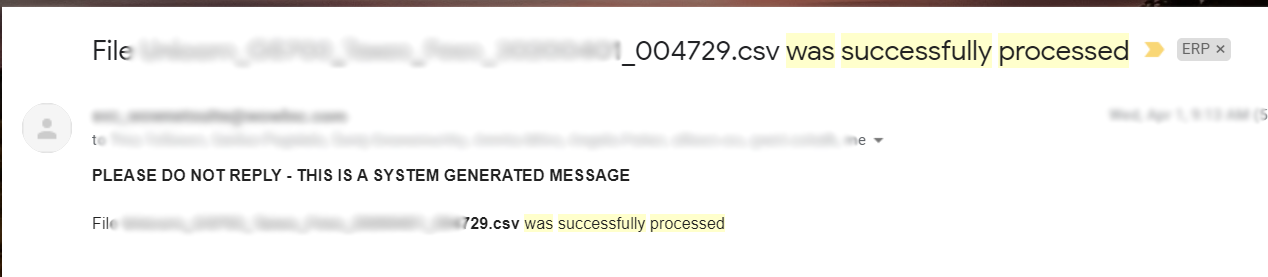
There are four options for invoking child pipelines:

* Pipeline Execute
* For Each – deprecated
* Task Execute – deprecated
* Nested Pipelines – deprecated, not supported

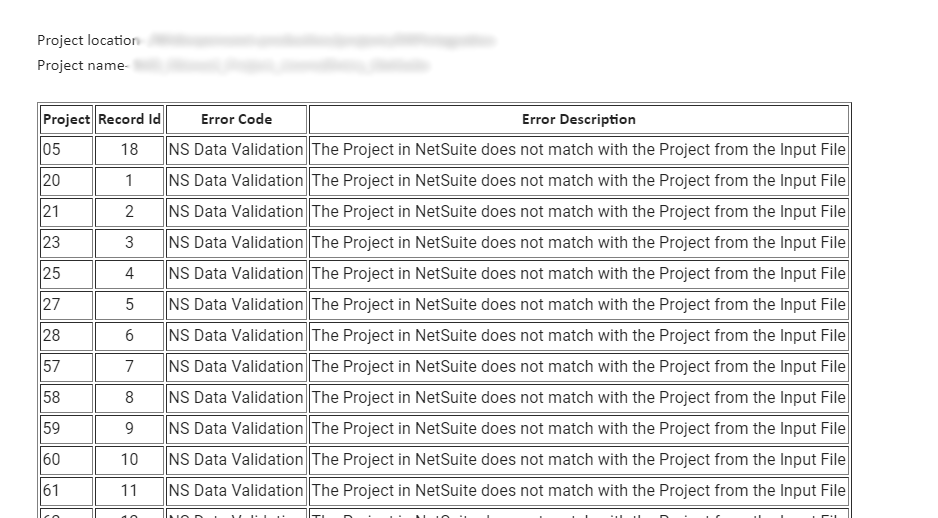
All options are deprecated except Pipeline Execute, with For Each snap being an exception and it is used in scenarios when asynchronous call to the child pipeline is needed. More detailed explanation of the Pipeline execute snap is available at official SnapLogic [documentation](https://docs-snaplogic.atlassian.net/wiki/spaces/SD/pages/1438684/Pipeline+Execute). Further, when using the pipeline execute snap, our practice and recommendation is populating the Execution Label option, for better insights and tracking in the Dashboard.

# Notifications

Notifications are an important part of the software development process. Often there is a requirement for sending notifications for events such as process started, process finished or when error occurs. Errors could be: Logical, Validation, System etc. However, errors should be tranSLated into friendly format and communicated with the users.



*Figure 31 – Example of email message*



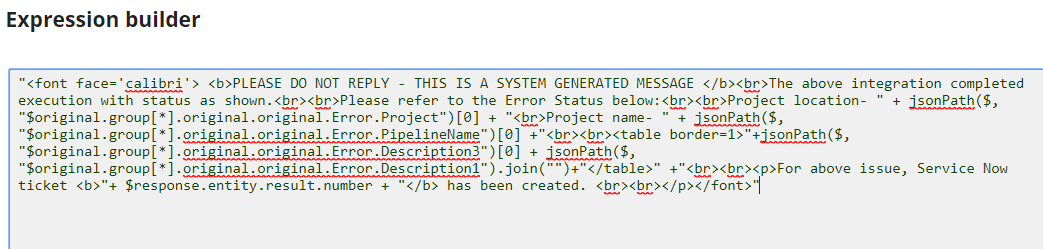
*Figure 32 – Example of errors summary*

## 8.1 Building email notification that contains table

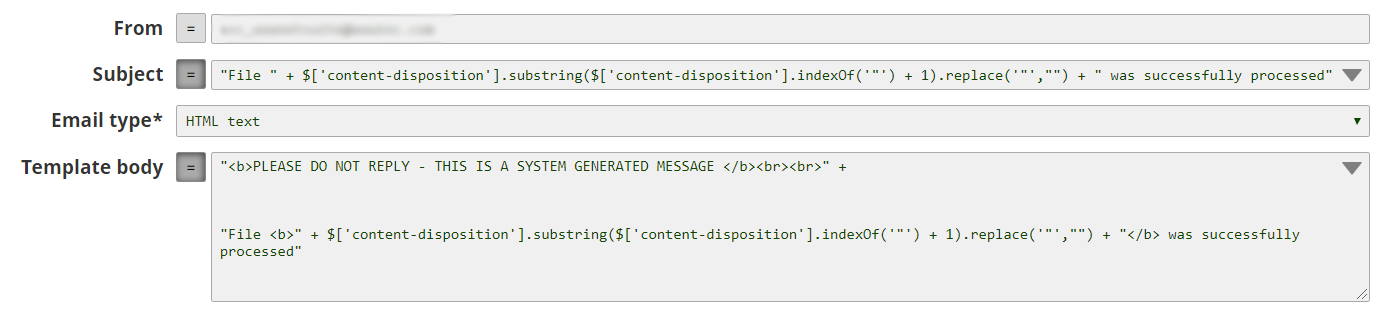
Creatign a table inside the email notification is handly and can be done in few way.

* + - 1. Building the table in the expression builder.

Can be used is some situations when tabel has a simple structure.



*Figure 33 – Building email body in with expressions*



*Figure 34 – Configuring of email sender snap*

Email notification with Email Sender snap is mostly used. This snap can be configured to send email messages to one or more recipients. Default behavior to send new email messages for every record in the input view. Email sender snap can be configured to group all records from the input view in one HTML table and then send one email message. Email Sender snap has the option for attaching files as well. Go to the official SnapLogic [documentation](https://docs-snaplogic.atlassian.net/wiki/spaces/SD/pages/1438208/Email+Sender) for detailed info.

* + - 1. Building the table using XSLT snap

Should be used when we have to build more complex tables because all the logic for building the table is placed in one file and it makes it easier to maintain.

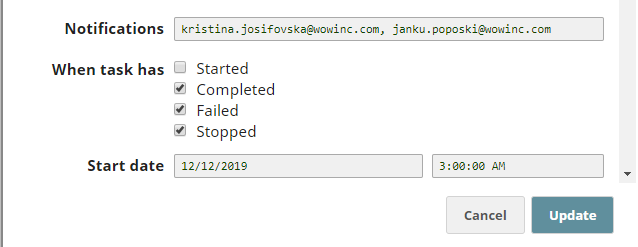
Following is a sample of an error email notification built using the XSLT file.



XLST pattern location:

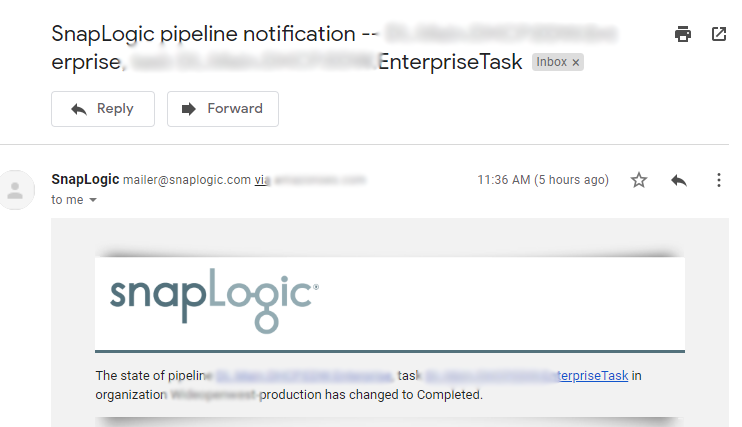
## 8.2 Task and Snaplex notifications

Tasks and snaplexes have built in automated notifications. Notifications field shown on Figure 35 can be filled with one or more comma separated email addresses of the recipients.



*Figure 35 – Task notification settings*

The notification email message contains pipeline and task names in the subject. Email body also contains task and pipeline names, the status of the pipeline and environment in which it is executed. Executed pipeline can be directly accessed from the notification email message.



*Figure 36 – Example of pipeline notification email*

general do’s and dont

# TRIGGER VS ULTRA TASK

**Ultra vs Triggered Task**

Both Ultra and Triggered are essentially similar types of tasks in SnapLogic, which means in both cases pipelines are exposed as APIs and can be invoked externally with a specific URL and basic authentication, enabled with Bearer Token. The main difference is that Ultra Tasks are created for specific Ultra Pipelines and they provide a high-availability and low-latency request/response API.

**Ultra Pipelines** have specific design and allow 10 times faster data delivery, enabling reliable, real-time completion of time-sensitive enterprise tasks and processes. Ultra Pipelines are always active and assure the delivery of data requests, which means if a request fails, it is picked up immediately by another instance and restarted without performance delays. Ultra Pipelines are executed as Tasks, which enable Pipelines to consume documents continuouSLy from external sources. When Ultra Pipelines are exposed as Tasks (**Ultra Tasks**), they can consume documents from sources not compatible with Triggered Tasks or those that require low-latency processing of their inputs. So, Ultra Tasks can manage multiple instances (runs) of a Pipeline and can consume documents simultaneouSLy to attain load-balancing/reliability.

Important to note is that, you can invoke the Ultra Task by using the URL that was created, but in cases where IP Address and port number are used as part of the URL, you must replace them with the name of the Host before you can invoke the Task.

When invoking the Ultra Task, authorization details must be provided. The authorization is enabled by a Bearer Token that can be sent in the authorization header or as a parameter when doing the request to the Ultra Task.

**Triggered Tasks**

Pipelines can be exposed as Triggered Tasks which means they can be executed by accessing a specific URL. When creating a Triggered Task there is an option to be included HTTP Basic Authentication by providing a Bearer Token. If the authentication with Bearer Token is enabled, the Bearer Token should be included in the authentication header when doing the request, or this token can be sent as a **bearer\_token** query parameter in the call.

\*\*\*When you are creating a triggered task, the Bearer token is prepopulated with a token. You can either trigger this tokenized Task through a Cloudplex or Groundplex, typically for use with a service account user. Clearing this field disables token authentication; authentication must then be done with a username and password. With Ultra tasks, an empty Bearer token means no authentication. \*\*\*

Pipelines that are exposed as Triggered Tasks don`t have some specific design such as the Ultra Pipelines, but they should have at most one open input view and at most one open output view. These kinds of tasks don`t provide the same performance features as the Ultra Tasks, so they can be used in scenarios where we don`t need essentially a real-time integration and low-latency processes. However, in order to obtain a better performance for the Triggered Task, you should follow the suggested practices for optimizing the Pipeline flow.

When you have a groundplex it is recommended that a triggered Task is invoked through an On-premise or Override URL because the Task is cached for subsequent executions. If we use On-premise URL the request the first time must come to the cloud control plane to fetch the metadata, but every other time the metadata is cached and we have no call to the control plan. The cached Tasks are then used on subsequent triggers of the URL, reducing task execution latency. Triggered tasks expire from the cache after an hour.

**Pipeline execute in Ultra mode**

Ultra pipelines must contain only snaps that are compatible with the ultra feature, in order to use the snaps that are not compatible we use the child pipeline approach with pipeline execute. With this approach we call a second pipeline (child) that is not in ultra mode and this pipeline is not in a memory. When the Pipeline parameter is hardcoded as in the screenshot below, the initialization of the child pipeline is very fast (in milliseconds) because the ultra pipeline knows what pipeline will be called every time but when the expression option is turned on than the initialization takes more time.

When using the other options in pipeline execute like reuse execution or pool size they act exactly the same as a normal pipeline but we need to be careful when using them because for example if we increase the pool size and the child pipeline has a snap that connects to some database then this approach will open multiple connections simultaneouSLy.