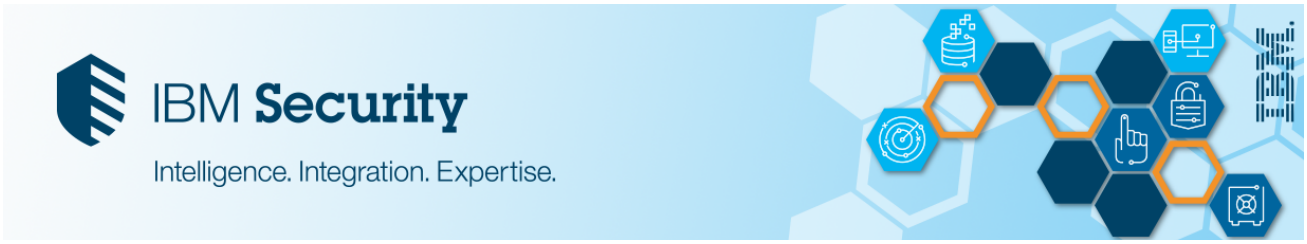


IBM SECURITY IDENTITY GOVERNANCE AND INTELLIGENCE

Rules Programming to Implement Custom Scenarios in IGI

5.2.x

David Edwards

Version 0.3
January 2018

Document Purpose

This document is a technical guide to using Rules to implement custom scenarios in IBM Security Identity Governance and Intelligence (IGI). Rules are an area that is often used in Proofs of Concept or deployments to extend IGI past standard configuration to implement specific customer requirements. This guide provides an overview of the use of Rules in IGI and the conventions associated with them, the queue-related rules for enhancing data flows, and other uses of rules in IGI. It includes extensive documented examples, some shipped with the product and some developed for specific scenarios.

The document assumes familiarity with IGI concepts, modules and functions.

Thanks to the following people for assistance with content or review:
- Fabrizio Petriconi, Senior IGI CTP, Italy
- Dmitri Chilovich, Cross-US Pre-sales Team US
- Rajesh Kumar Baronia, AP IAM Pre-sales, Singapore
- Alberto Novello, IGI Development, Italy
- Fabrino Calvarese, IGI Development, Italy
- Gabriel Rebane, Identity CTP, US
- Rafik Mezil, Identity CTP, France
- Vaughan Harper, IAM CTP, UK

For any comments/corrections, please contact David Edwards (davidedw@au1.ibm.com).

Document Conventions

The following conventions are used in this document:

A note, some special information or warning.

```
A piece of code
```

Normal paragraph font is used for general information.

The term “IGI” is used to refer to IBM Security Identity Governance and Intelligence.

Document Control

Release Date	Ver	Authors	Comments
08 Nov 2017	0.1	David Edwards	Initial draft version (written against IGI 5.2.3.1)
21 Nov 2017	0.2	David Edwards	Minor updates and extra examples from peer review
Jan 2018	0.3	David Edwards	Various code examples added

Table of Contents

1 Using Rules in IGI to Extend Product Functionality

IBM Security Identity Governance and Intelligence (IGI) provides a number of mechanisms to extend the out-of-the-box configurable functionality to implement customer requirements. These include reports that can query on almost every record in the IGI database, extensible workflow processes and activities, and rules that can be invoked at many points throughout the product. This document focusses on the Rules and how to use them to extend IGI.

1.1 Introduction to Rules

IGI uses Rules to allow deployments to extend product functionality. The following sections introduce the concept and implementation of Rules in IGI. Later sections of the document provide more in-depth explanation of rules.

1.1.1 What Are Rules?

As with many products, IGI exposes mechanisms to extend and enhance product functionality. Rules are a programmatic way to do this, leveraging the Java programming language, the Drools Java engine, APIs and IGIs data model.

Rules provide a trade-off between high flexibility and effectiveness, a steep initial learning curve, and a moderate level of maintainability, against having the product support every unique use case via UI configuration options. Often commonly-used rule functions become product configuration options (like email notifications and triggering workflows from campaigns).

Rules provide a means to alter product behaviour without compiling, destroying or restarting processes. Rules are self-contained Java modules, implemented in the Drools engine via the IGI Admin Console. They do not need to be compiled, nor do they need to be deployed as JAR/EAR files into the Java directory structure on the Virtual Appliance – they are stored internally by IGI.

Rules are effective, but require practice and experience, creativity and documentation (such as this document).

The most common use of Rules is to alter the data or flows for events coming into IGI (like user changes flowing from a HR system, or accounts and access rights from a reconciliation) or flowing out from IGI (provisioning account/access changes). They are also often used to perform complex data mapping for adapters and connectors. They can also be used to customize behaviour for certification campaigns, building complex hierarchies, workflow processes and activities, custom policy (like userid creation) and other functions.

The Drools Java Engine

A rules engine (RE) is a module that automates the management of certain highly variable processes. The fundamental concept consists of separating the objects that are involved in processes from the logic that implements those processes.

The logic is defined by writing rules. For each process, the RE recognizes which rules to apply and on which objects to operate. If there is a variation in the logic, the rules can be changed without having to intervene in the system architecture. The IGI framework uses the open source Drools Rules Engine (www.jboss.org/drools), which enables the properties and advantages mentioned.

The rules engine is largely transparent to the use of Rules; you create and edit Rules in the IGI Admin Console and the Rules Engine manages and executes them. Rules are compiled as needed (and there is a rules cache for performance).

Drools has its own syntax for writing rules in a declarative, concise, and unambiguous format. A rule has the following structure:

```
when
   conditions
then
   actions
```

The *conditions* are normally a set of Java beans in the working memory that relate to the operation. If the beans are found, the rule is executed. You need to be careful to ensure that there is logic to handle null beans (for example, you might have a rule processing account and user beans, but an orphan account will have a null user bean). The deeds in the conditions section can include evaluation of bean attributes.

With IGI 5.2.3.1, the conditions (deeds in Drools terminology) are pre-filled for a given rule type.

The *actions* are normal Java code to implement the required business logic. They will normally involve beans (representing the objects managed by Access Governance Core) and actions (representing the actions that can be executed on these objects through the AG Core administration module). Each rule has objects that are available in the working memory only. The Action methods can be applied on these objects only.

You should read the information on the Drools implementation at https://www.ibm.com/support/knowledgecenter/en/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/RULES_ENGINE/RUD_DroolsRulesEngine.html. It provides a more technically precise explanation of the conditions (deeds and working memory) and syntax (expression of beans).

Rules vs. APIs

IGI provides a number of application programming interfaces (API's) including a Java EJB implementation and multiple REST APIs. https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/reference/cpt/cpt_api.html.

Remote application programs run outside of the IGI JVM. Classes outside of the application packages are not intended to be started by a remote application. Classes in remote applications are documented under the IGI application packages. Server extensions, which run in the IGI JVM, can use any of the classes that are listed in the published API documentation (Javadoc). They are Java classes that run in the same JVM of the caller. These APIs are used to develop Identity Governance and Intelligence customization and extensions that can plug into Identity Governance and Intelligence.

The Rules will leverage the Java EJBs from within the IGI JVM, run by the Drools rules engine.

1.1.2 Rules, Rules Flows and Other Terminology

Rules are a single module, comprising the *when – conditions – then – actions* block described above.

Rules are contained in a Rules Sequence (also known as Rules Flow). A Rules Sequence, or Rule Flow, is a collection of rules. New rules sequences can be created for some types of rules.

Sequences of rules are grouped in Rule Classes. Rule classes map the rules to specific modules and functions. Some classes of rules (system rules classes) have some fixed sequences, as shown in the following table:

Rule Class	Module	Fixed Flow (Sequence)	Triggered by
Live Events	Access Governance Core	Yes	Events execution (with exception of creation events)
Deferred Events	Access Governance Core	Yes	Deferred Events execution (with exception of creation events)
Authorization Digest	Access Governance Core	No	Fulfilment of a certification campaign
Advanced	Access Governance Core	No	Not triggered but can be scheduled by a job of Task Planner
Account	Access Governance Core	No	Account creation
Password	Access Governance Core	No	Password policy
Attestation	Access Governance Core	No	Creation of a data set for a certification campaign
Hierarchy	Access Governance Core	No	Building of an attribute hierarchy
Workflow	Process Designer	No	Pre-action or post-action that is related to a workflow activity
Advanced	Access Risk Controls for SAP	Yes	SAP system operation

The rules can be categorized into two groups; rules tied to event flows (with the queues) such as the Live Events and Deferred Events rule classes, and the other rules tied to specific functions or operations within IGI. Not shown in the table are the mapping rules tied to Enterprise Connector definitions (related to event flows).

Rules tied to event flows operate on data flowing into, or out of IGI. This is user/people changes from a HR system, account/permission changes flowing from a reconciliation, or account/permission changes being provisioned to a target. These may be used for data mapping or additional activity based on data events. A good example is when responding to a new account from a reconciliation, rules can be used to programmatically match the account to an existing user by userid, email, surname or first name.

Other rules can be applied to specific functions in IGI, such as building a userid for a new person/account, responding to a REVOKE action in a certification campaign, running a background task, adding pre- or post-processing to steps in a workflow process or rebuilding a complex attribute hierarchy.

These will be explored later in this document.

1.2 Rules Documentation

The documentation on rules has been fragmented and limited, thus this document. This section summarizes where to find the existing documentation.

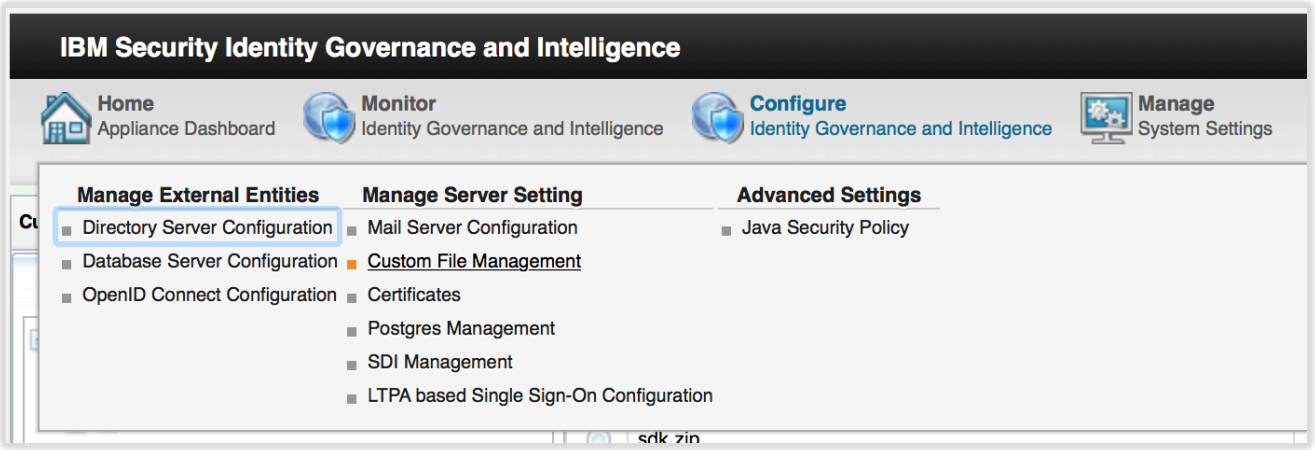
1.2.1 Accessing the Object/Method Documentation

IGI ships with a Software Development Kit (SDK) that includes:

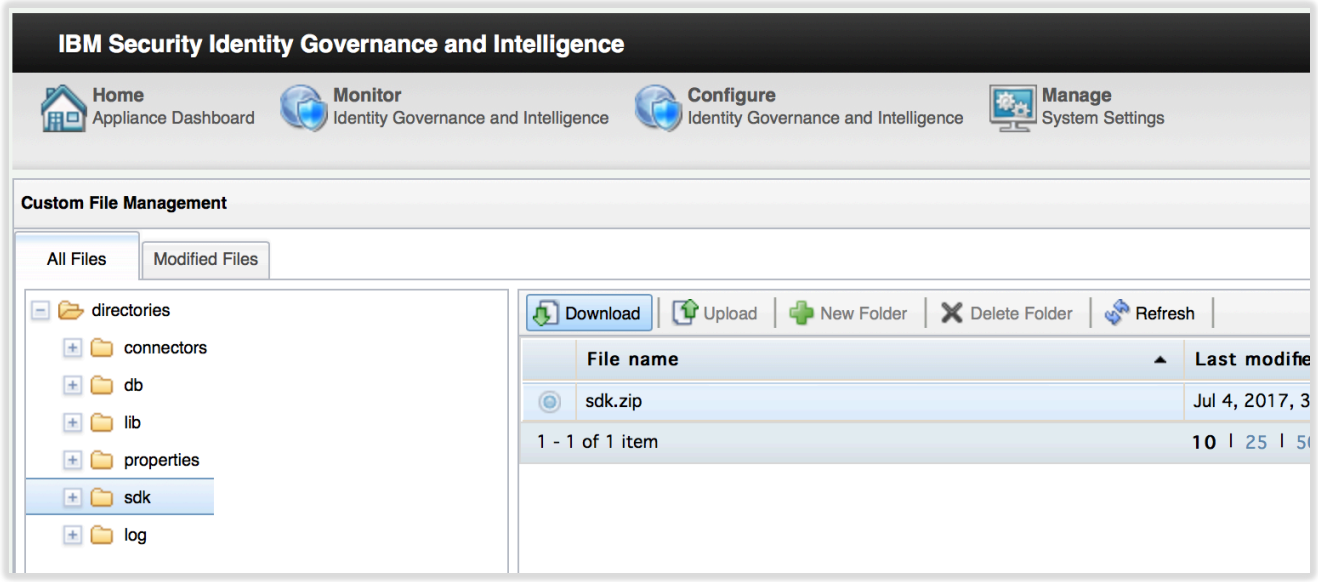
- *customization* - Files used to customize Identity Governance and Intelligence. For example, adding a custom application in the desk, changing the labels and descriptions of the applications, and setting the date and time format for the entire product.
- *javaDocAGCore* - The Javadoc, which provides the documentation for the IGI EJB.
- *lib* - The binary versions of the IGI libraries and WAS client to compile the SDK source.
- *Readme* - A README.txt file.
- *RESTDoc* - Documentation to create REST API calls to the IGI services.
- *RESTExamples* - Examples of the REST API calls.
- *src* - The source code of the SDK.
- *sas.client.props* - The WebSphere Application Server access configuration information.
- *ssl.client.props* - The SSL information.

Only the javaDocAGCore content is relevant for rules coding.

The SDK file is downloaded from the IGI Virtual Appliance Local Management Interface (LMI). It can be found under **Configure** Identity Governance and Intelligence > Custom File Management

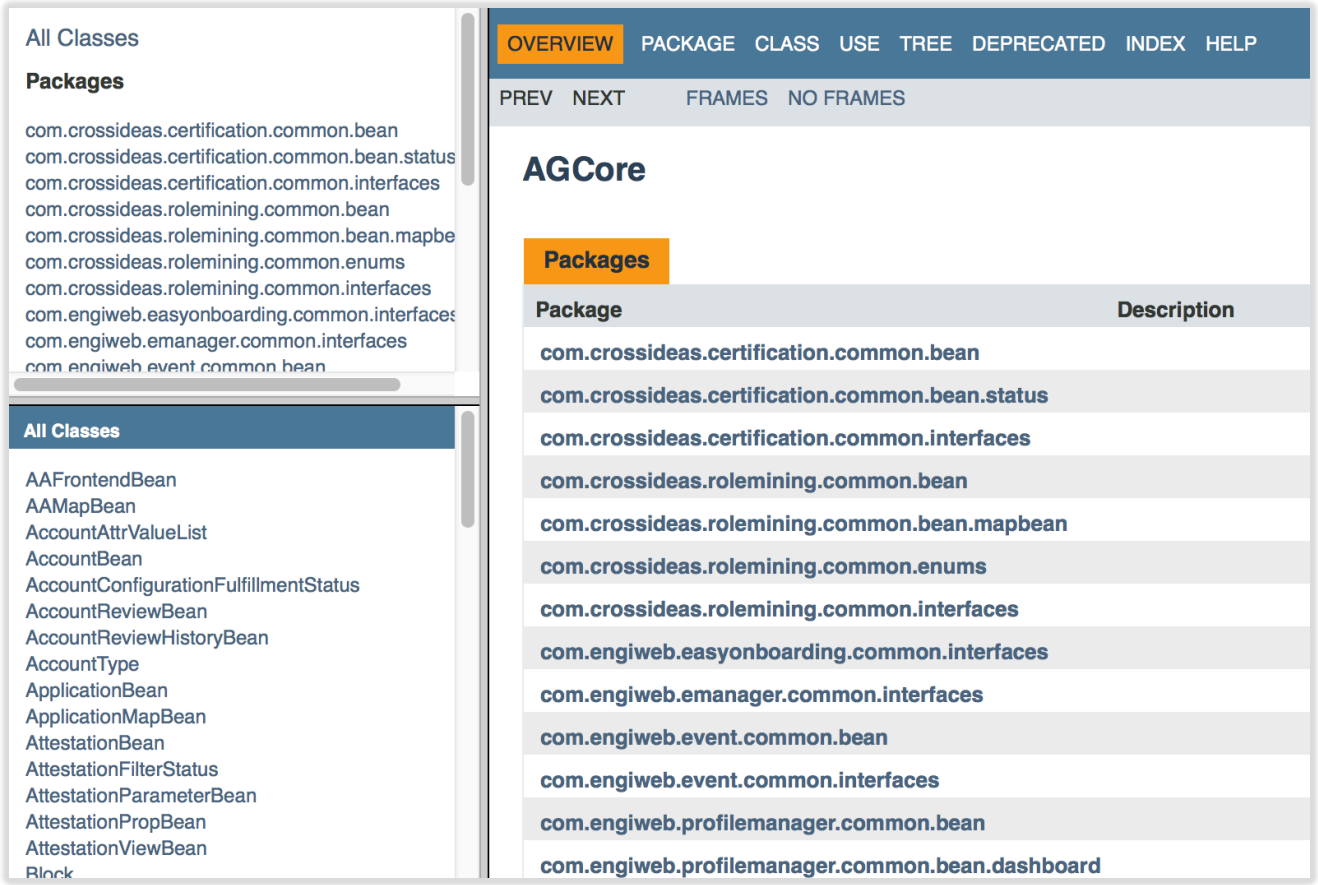


The zipped SDK file is under the sdk folder.



To download, select the sdk.zip file and Download. The browser will download the file in the normal way.

Uncompress the zipped file to access the folders listed above, including the javaDocAGCore folder. The index.html file provides browser access to the Java Docs.



The ******Bean*** classes are the most relevant to rules coding.

1.2.2 Sample Rules Shipped with IGI

There are many sample rules shipped with IGI. They can be accessed via the Configure > Rules tabs in the Access Governance Core, Access Risk Control for SAP and Process Designer. Many of these will be used in this document.

1.2.3 Rules Documentation

Information on Rules is scattered throughout the official product documentation, including:

- A Rules introduction – https://www.ibm.com/support/knowledgecenter/en/SSGHJR_5.2.3.1/com.ibm.igi.doc/administering/cpt/cpt_ac_rules_introduction.html
- AGC Rules – https://www.ibm.com/support/knowledgecenter/en/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/AGC/rulez.html
- Process Designer Rules – https://www.ibm.com/support/knowledgecenter/en/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/PD/GestioneRules.html
- ARCS Rules – https://www.ibm.com/support/knowledgecenter/en/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/ARCS/GestioneRules.html
- Reference – Rules Overview – https://www.ibm.com/support/knowledgecenter/en/SSGHJR_5.2.3.1/com.ibm.igi.doc/reference/cpt/cpt_rules_reference_overview.html

This documentation (5.2.3.1) is significantly enhanced over previous iterations of the documentation and we expect it will improve with future iterations of the product documentation.

1.3 Supporting Knowledge for Effective Rules Development

To be able to write rules to implement business logic into IGI, you should be familiar with the following IGI topics:

- The IGI Data Model – the objects and their relationships in IGI
- IGI Data Flows – particulalry the use of the Queues (IN, OUT, TARGET, INTERNAL) and the types of data the flows and the operations on those data types
- Processes and Activities used in Access Request Management – how user workflows are implemented into the IGI Service Center
- Certification Datasets and Campaigns – for re-certifying access
- Tasks and Jobs in the Task Planner – the background activities run on schedules
- Account Management – including userids and password management

All of these topics are covered in the basic IGI enablement, available online or via face-to-face classes.

2 Rules Conventions and Common Practices

This chapter covers some of the common aspects of rules development and implementation.

2.1 Understanding Rules, Sequences, Classes, Flows and Packages

A rule is a discrete piece of business logic coded in Java using the Drools rules engine (following the *when – conditions – then – actions* convention) associated with an event or operation in IGI. For example, it could be some business logic associated with a USER_ADD event on the IN queue, or some logic to rebuild an attribute hierarchy or process a Revoke action in a campaign.

Thus, a rule must be associated with an event or operation in IGI, and this association involves rule classes, queues (optionally), rule flows and rule packages. These, and how they are related, are described in the following sections.

2.1.1 Rule Concepts

This section describes the concepts behind rule management.

Rule Classes

Rule classes are high-level categories of rules, that associate rules with IGI modules and flows/sequences (and optionally queues).

Rule Class	Module	Fixed Flow (Sequence)	Triggered by
Live Events	Access Governance Core	Yes	Events execution (with exception of creation events)
Deferred Events	Access Governance Core	Yes	Deferred Events execution (with exception of creation events)
Authorization Digest	Access Governance Core	No	Fulfilment of a certification campaign
Advanced	Access Governance Core	No	Not triggered but can be scheduled by a job of Task Planner
Account	Access Governance Core	No	Account creation
Password	Access Governance Core	No	Password policy
Attestation	Access Governance Core	No	Creation of a data set for a certification campaign
Hierarchy	Access Governance Core	No	Building of an attribute hierarchy
Workflow	Process Designer	No	Pre-action or post action that is related to a workflow activity
Advanced	Access Risk Controls for SAP	Yes	SAP system operation

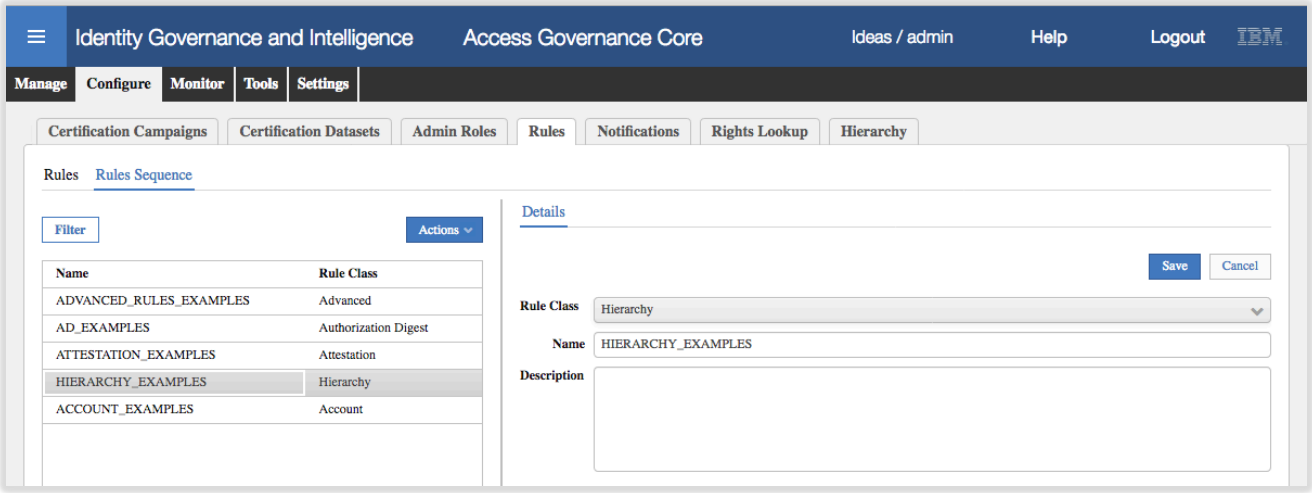
Every rule will be in a class, and that class will define what queues/flows the rule can run against. The exception to this are the rules that can be associated with Enterprise Connector definitions (covered later).

For example, rules that run against events flowing into or out of IGI via the Queues will normally be in the Live Events class, which means they can be associated with one of the four live events queues (IN, OUT, TARGET or INTERNAL) and each of the queues will have a fixed set of operations (defined as Rule Flows) the rule can be applied to, such as USER_ADD.

Other rules may be associated with operations in IGI will be associated with different classes, not involve queues and do not have a pre-defined set of flows/sequences. For example, a workflow rule is in class Workflow, and belong to a custom flow/sequence.

Rule Sequences

Rule Sequences are low-level categories for rules that can be used for some rule classes. Any rule, other than AGC event rules and ARCS rules, can be associated with a rule sequence. A specific flow can only use rules from the same rule sequence.



The figure above shows a set of rules sequences in Access Governance Core for different rule classes. The one highlighted is HIERARCHY_EXAMPLES – a Hierarchy class container for hierarchy build rules.

Queues

Queues are the external interface mechanism for IGI – external system will put events on queues to push data into IGI, and pull events off queues to process data from IGI.

To associate a rule with an event type in IGI, they must be associated with a queue. There are a number of queues currently available for rules processing:

- Live Events/IN – Processing incoming user and org unit events (e.g. HR Feed)
- Live Events/OUT – Processing outgoing account and permission events (aka provisioning)
- Live Events/TARGET – Processing incoming account and permission events (e.g. reconciliation)
- Live Events/INTERNAL – Processing some internal events
- Deferred Events/IN – Processing incoming user and org unit events that have been deferred

There is an additional queue introduced in 5.2.3.1 for outgoing user events, primarily implemented for ISIGADI (ISIM-IGI integration).

The other rule classes do not use queues to associated rules to operations.

Rule Packages

Rule packages are libraries of rules that can be attached to a Rule Flow/Sequence. It is a set of all the rules that could be applied to a specific operation. Consider it a library of rules. You could have a collection of rules that could be used against a user add operation or a collection of workflow rules.

Rule Imports

Rule imports are the header for the rule, with the import and global statements for the code. Each bean or object used in the rules must have an associated import statement in the header. This becomes relevant if custom packages are added to IGI.

"Run Rules" or Rule Flow

The run rules (sometimes referred to as the rule flow) is the sequence of rules associated with an operation. Rules will be processed in sequence, top to bottom, and will need to include routing logic if you want a specific rule to not execute based on a previous rule execution.

The rule flow may be one of the defined event types (e.g. USER_ADD with Live Events/IN) for event-related operations, or a custom rule sequence for the non-event rules.

Note that a rule flow may be a discrete set of rules, or rules may be collected in a folder in the flow. Either way the rules will be processed in sequence, top to bottom.

2.1.2 Rule Concept Examples

This section provides some examples of how the concepts are applied.

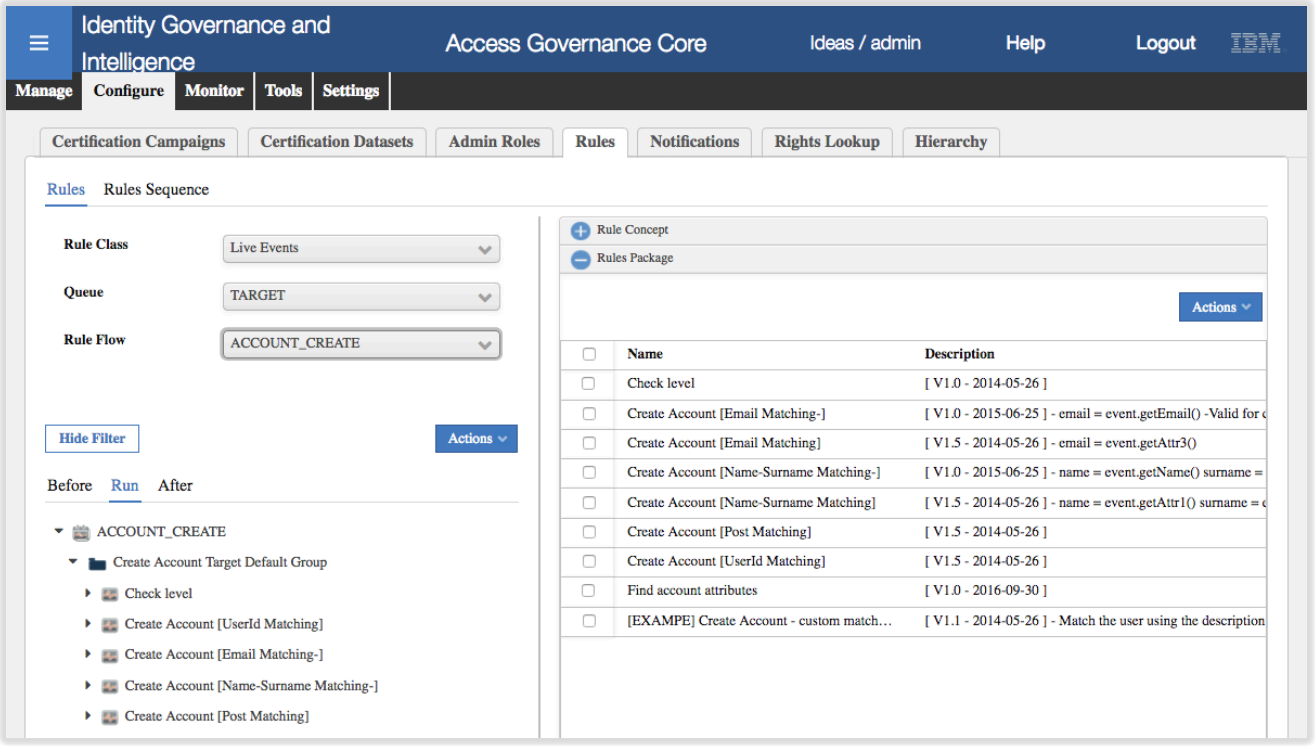
Example: Account Matching Rules for New Account Events

An implementation needs to associate some business logic for account-user matching when a new account is found during a reconciliation.

In this case:

- The *rule class* is `Live Events` as we were operating on live account events flowing into IGI
- The *queue* is `TARGET` as it processes all account and permission events from targets, and
- The *rule flow* is `ACCOUNT_CREATE`, as that is the flow for a new account.

The *Rules Package* for the ACCOUNT_CREATE flow contains a library of rules that could operate on an ACCOUNT_CREATE event. New rules could be added to the package.



The actual flow ("Run") for an ACCOUNT_CREATE event consists of the following rules:

- Check level
- Create Account [Userid Matching]
- Create Account [Email Matching-]
- Create Account [Name-Surname Matching-]
- Create Account [Post Matching]

As you can see these are a subset of the Rules Package list. Note how the Rules Package mechanism can be used for versioning – you can hold multiple versions of a rule and apply them to

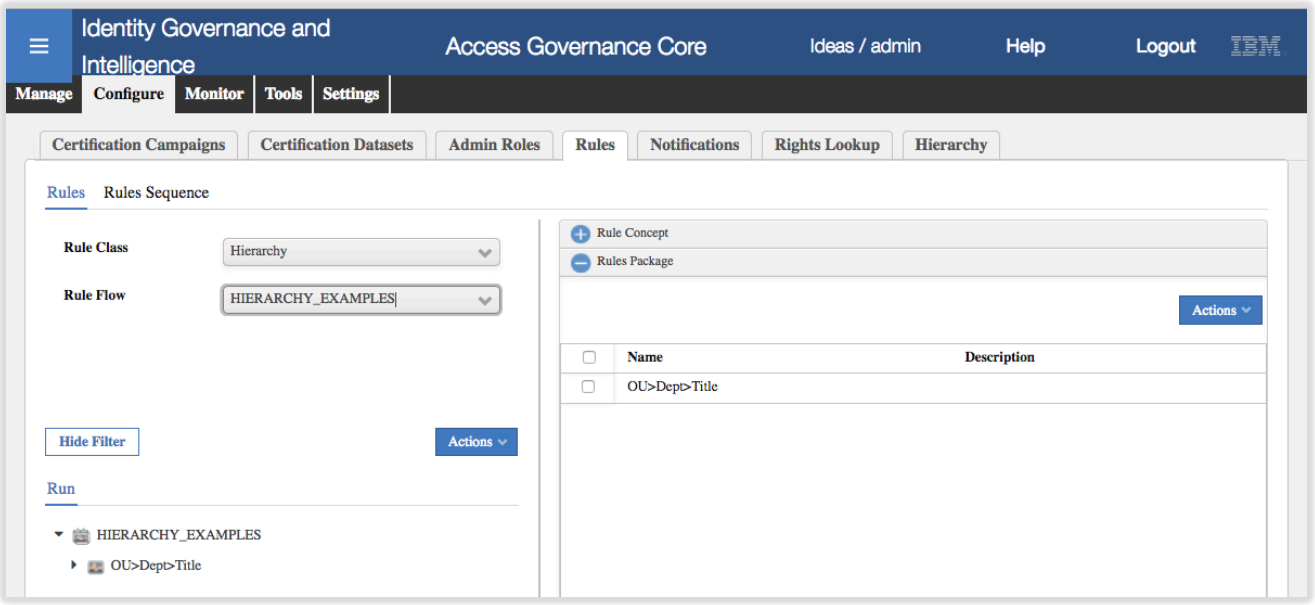
We won't explore these rules now, but they are a set of modules that will attempt to match this account to an existing user (by userid, then by email, then by names) and create the account. If no match is found, an unmatched account is created. This sequence of rules is run for every new account event (ACCOUNT_CREATE) that arrives on the TARGET queue in IGI. Note – ignore the Before and After tabs in the Rule Flow rules. They are no longer used.

Example: Hierarchy Build Rules

An implementation needs to have a job to rebuild a complex attribute hierarchy based on a user's OU, Department and Title. A rule is written to perform this.

As this is a non-event rule it must be associated with a custom Rules Sequence. A rules sequence called "HIERARCHY_EXAMPLES" has been created (see above).

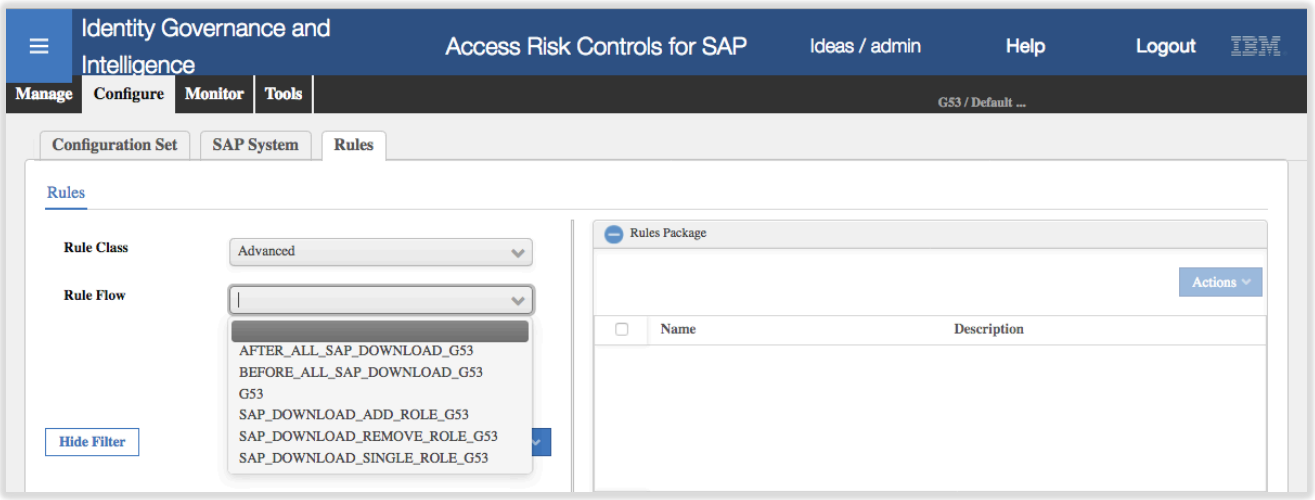
To enable the new rule, a *Rule Class* of `Hierarchy` is selected, then the custom `HIERARCHY_EXAMPLES` *Rule Flow*, the new rule added to the *Rules Package* and then associated with the Run for the flow.



The other non-event rules are implemented in the same way. They also need to be associated with their specific operation, such as a job or campaign, which will be covered in a later section.

A Note on ARCS Rule Flows

The Access Risk Controls for SAP (ARCS) module is unique in how it manages Rule Flows. Whereas AGC event rules are associated with queues and event types on queues, the ARCS flows are associated with SAP instances defined to ARCS and standard IGI operations for those instances.

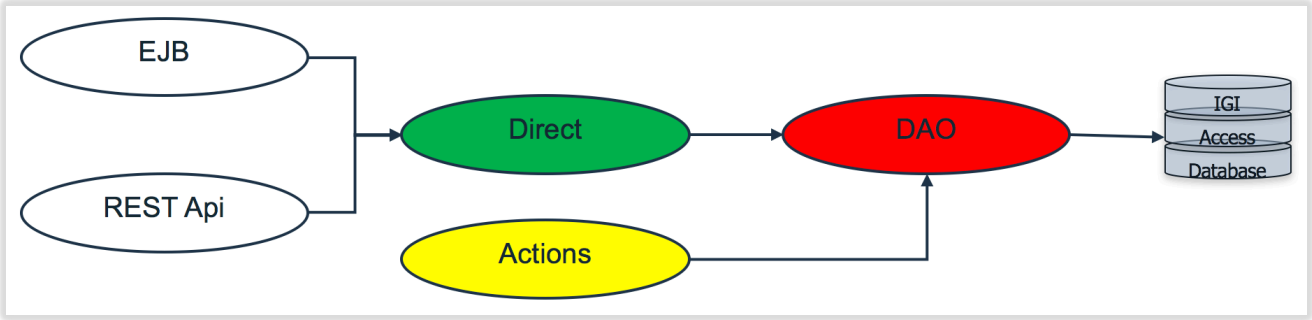


ARCS and ARCS rules are a very specialized area and not covered in this document.

2.2 Objects and Methods

The IGI product (or Ideas as it was under CrossIdeas) has a long heritage of extending the functionality through rules and external API-based programs.

The following figure shows the programmatic ways to access the IGI data objects and functions.



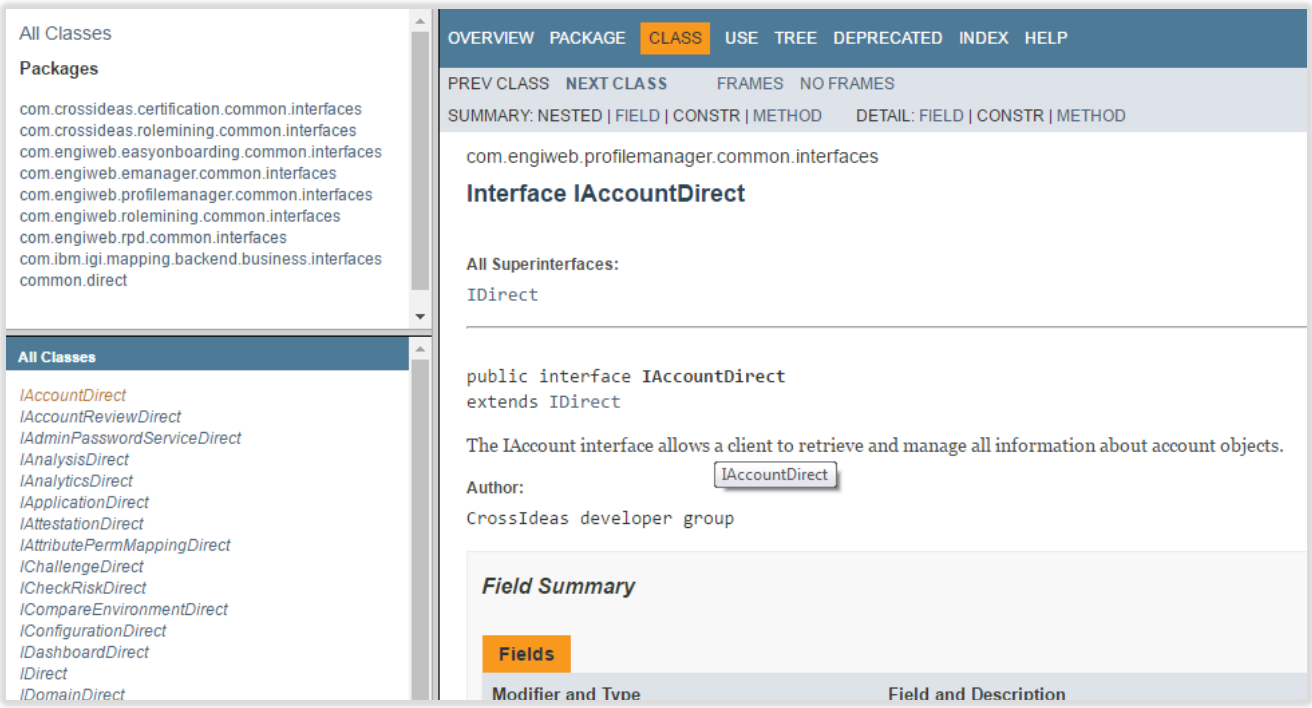
External APIs are provided in the form of Enterprise Java Beans (EJB) and REST APIs. The REST APIs have been significantly enhanced with IGI 5.2.3 FP1.

Both the EJB and REST API calls will leverage the Direct classes to call methods on the objects. The Data Access Objects (DAO) provide the database abstraction.

The Actions are business classes usually used by rules like Utilities. They will be deprecated just to have a unique approach inside IGI (use the Direct classes).

You should not use the DAO classes directly as they can change (IGI DAO classes are not an API). Note that you will see some examples in this document use the DAO classes as they are old examples that have been around for some time.

The Direct classes can be found in the JavaDoc with an "I" prefix (normally in italics).



An example of using a direct class is as follows:

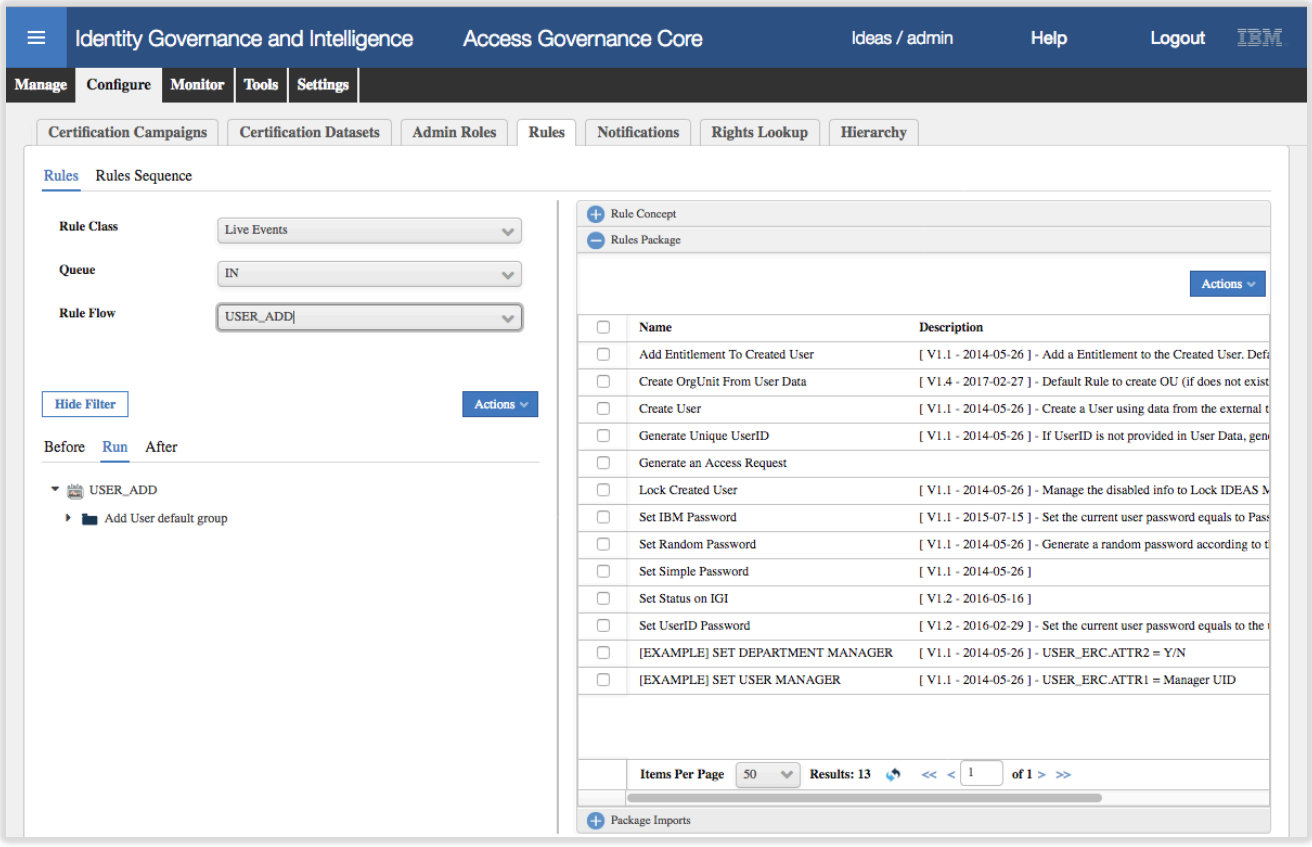
```
when
   event : EventTargetBean( )
   account : AccountBean( )
   attributes : AccountAttrValueList( )
then
// [ V1.0 - 2016-09-30 ]

   if(event.getTarget().startsWith("ILC_")) {

      AccountDirect accountDirect = new AccountDirect ();

      // Gets the attributes from ILC.
      BeanList<PwdManagementAttrValBean> beanList = accountDirect.findAttrValue(null, account, null, sql);

      attributes.addAll(attributes);
   }
```

To use a “Direct” class just instantiate it, then use its methods passing the required data. The methods always require a connection parameter: you need to pass the global variable “sql”.

There are many examples in this document, some use the direct classes, others use DAO classes. The examples are included to show how different outcomes can be achieved in the product, but you should endeavour to use the direct classes (or EJBs) wherever possible.

2.3 Managing and Editing Rules and Rule Flows

Rules can be managed in the IGI Admin Console, under the **Configure > Rules** tab in Access Governance Core, Access Risk Controls for SAP and Process Designer.

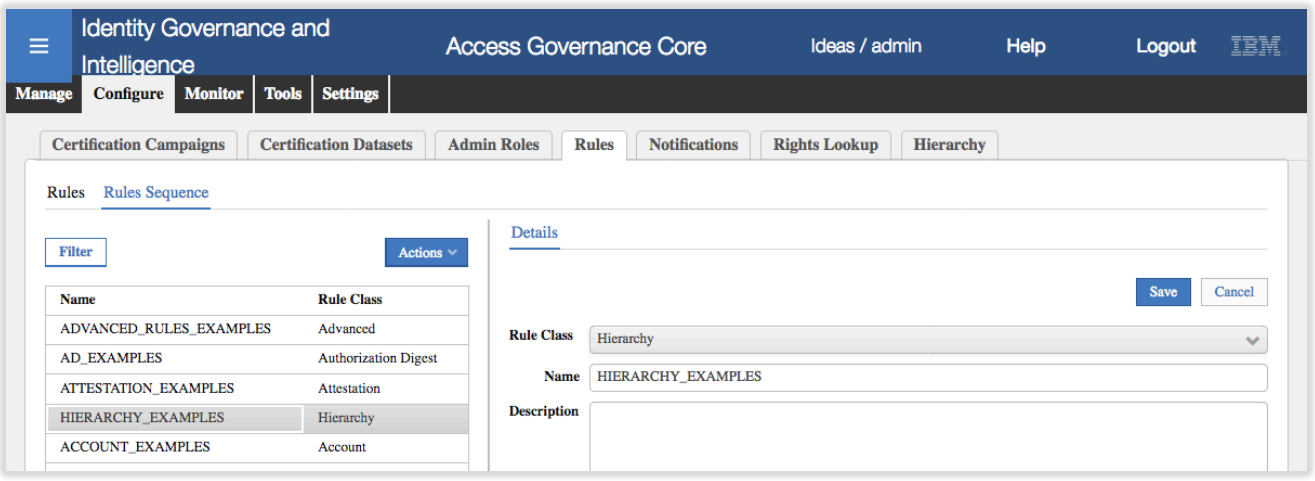


This interface allows for the management of rules sequences, rules packages, package imports and the rule flow execution. The following sections describe how the UI is used to manage each of the components.

2.3.1 Managing Rules Sequences

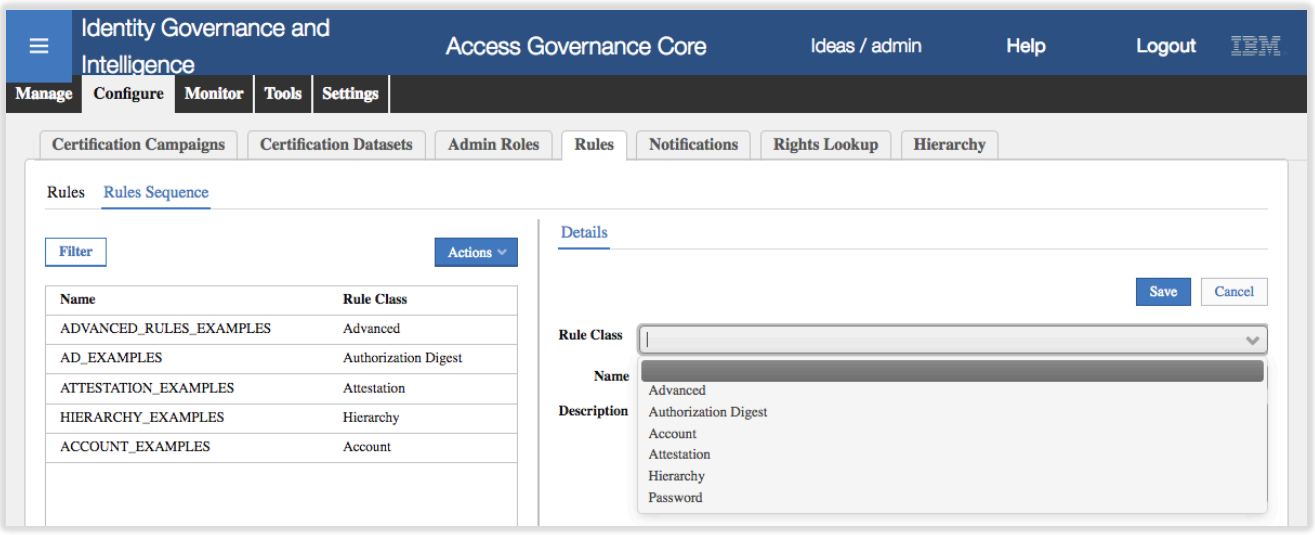
Rules Sequences are used to categorize rules for non-event rule classes.

The Rules Sequences can be managed by clicking the Rules Sequence link at the top of the rules page in the Access Governance Core or Process Designer (there are no Rules Sequences for ARCS).



From here you can create new rules sequences or remove old ones. You can have multiple rules sequences for a specific rules class, allowing separation of sets of rules.

Rules Sequences of class Workflow can only be managed in the Process Designer. Rules Sequences of class Advanced, Authorization Digest, Account, Attestation, Hierarchy and Password (for 5.2.3) can be managed in the Access Governance Core.

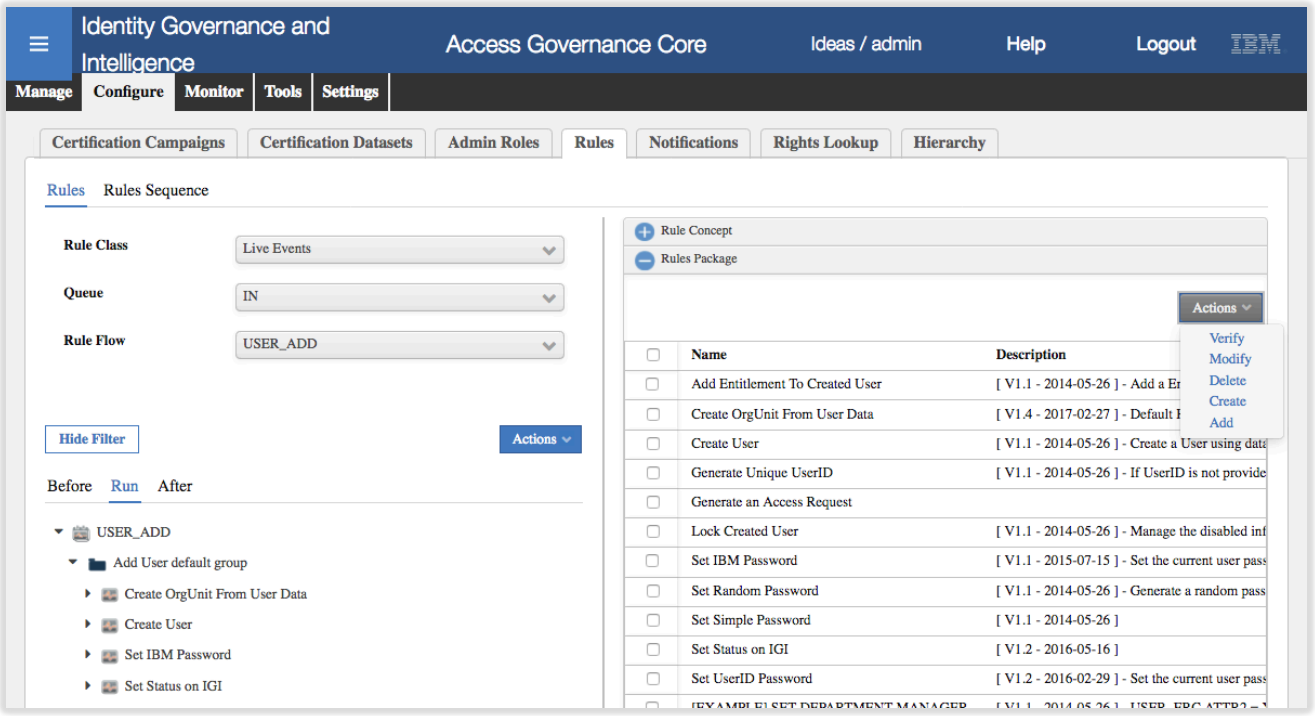


Note that you don't associate Rules with the Rules Sequences in this view. That is done in the Rules view.

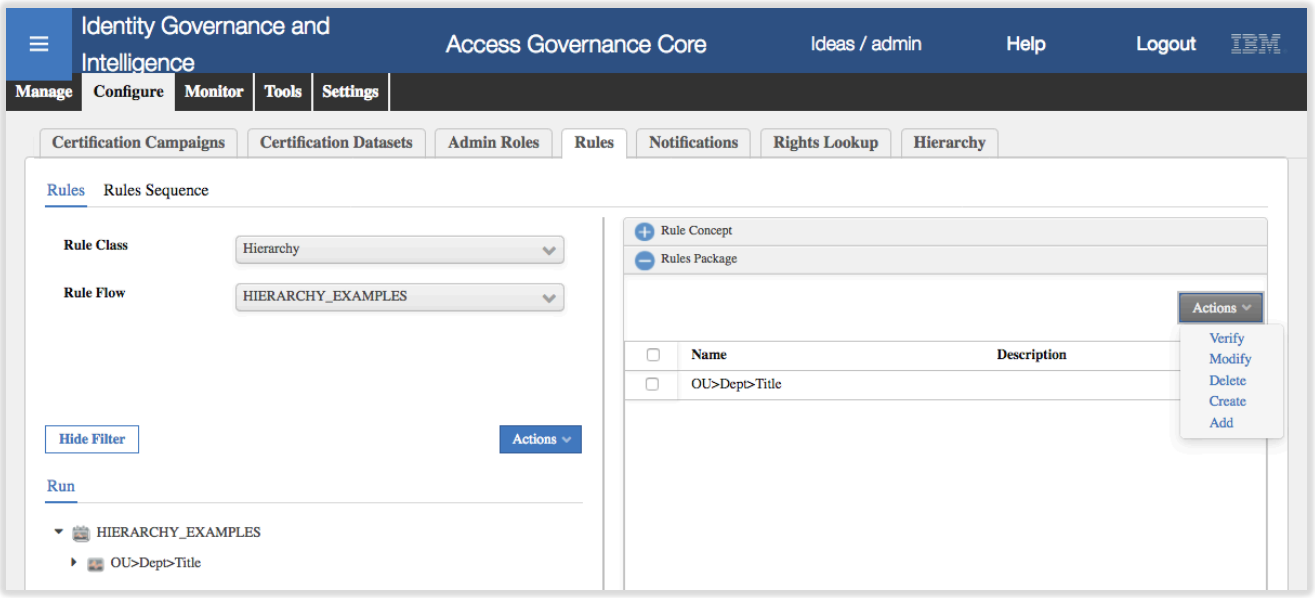
2.3.2 Managing Rules Packages

Rules Packages contain the set of rules that can be used to build a rule flow. They will be tied to either a specific queue/event for event-based rules (like IN/USER_ADD) or a Rules Sequence for a non-event rule.

Rules Packages are managed under the **Rules** link on the Rules page (in Access Governance Core, Process Designer and ARCS). For example, the Rules Package for Live Events/IN/USER_ADD are rules that can be used for USER_ADD events flowing through the IN queue.



Similarly, the Hierarchy/HIERARCHY_EXAMPLES Rules Package contains rules that can be used to hierarchy rebuild operations.



The Rules Package is a superset of rules used in flows. It can be used as a version control mechanism.

The following Actions can be used for Rules Packages:

- **Verify** - Checks the formal structure of the code that defines a rule (needs the selection of one of the rules listed). This is basically a syntax check and compile.
- **Modify** – Open the Rules Editor for editing a rule
- **Delete** – Deletes a selected rule
- **Create** – Creates a rule
- **Add** – Adds a selected rule to the current rule flow ("Run").

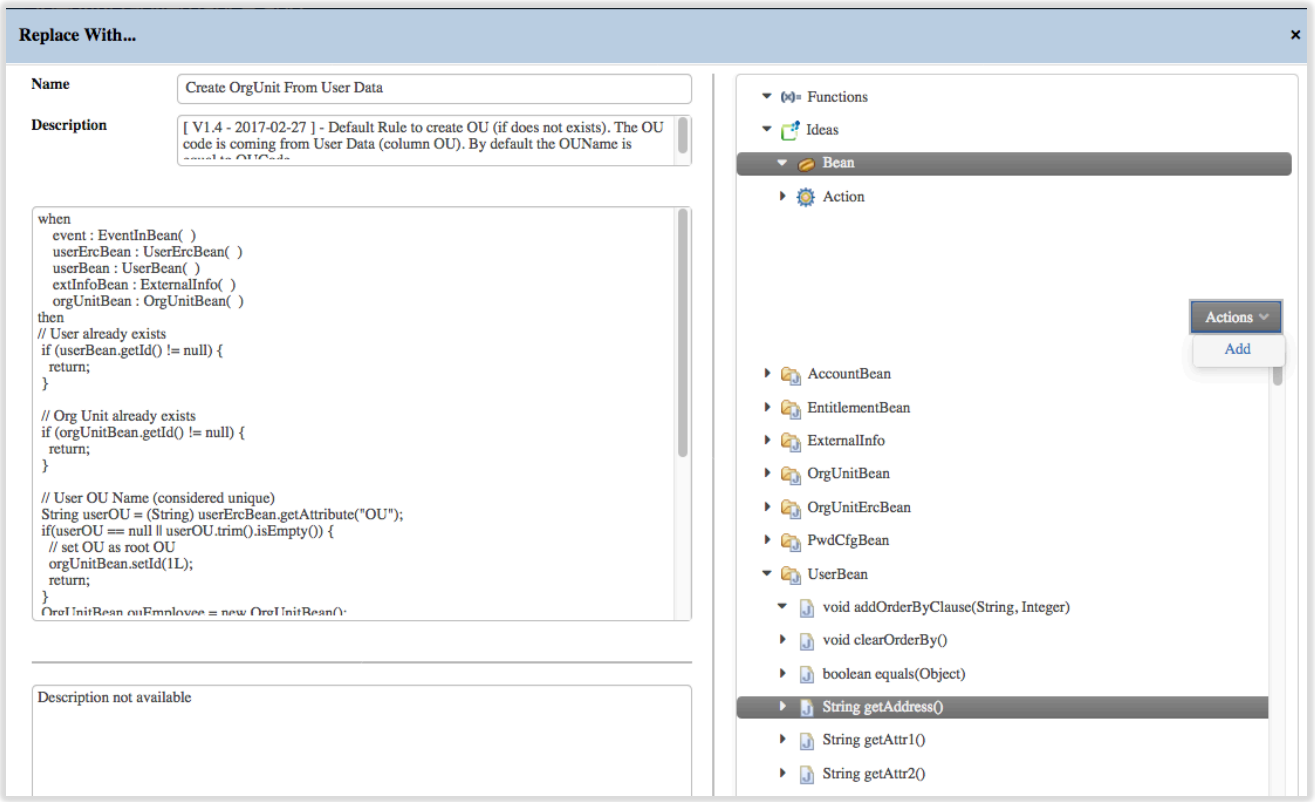
Thus, rules are put into, managed and deleted from Rules Packages. They are attached to a Rule Flow from the Rules Packages.

2.3.3 Rules Editor

The Rules Editor is started when a new rule is created or an existing rule is modified.

The Editor contains two panels:

- The left panel is for editing the current rule, with fields for the Name, Description and the Drools/Java code.
- The right panel is a helper for functions, beans, and actions that can be added into the rule



More information on the Rules Editor and helper objects can be found in https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/RULES_ENGINE/RUD_RulesPackage.html.

2.3.4 Managing Package Imports

The package imports contain the import definitions for all the classes that must be included for the rules. To find the package imports needed, you can check existing rules or look at the JavaDoc (unfortunately not all classes are in the JavaDoc yet).

If you add your own custom classes to IGI and want to reference objects in the rules, you must include the relevant import command in the Package Import.

2.3.5 Managing Rule Flow Execution

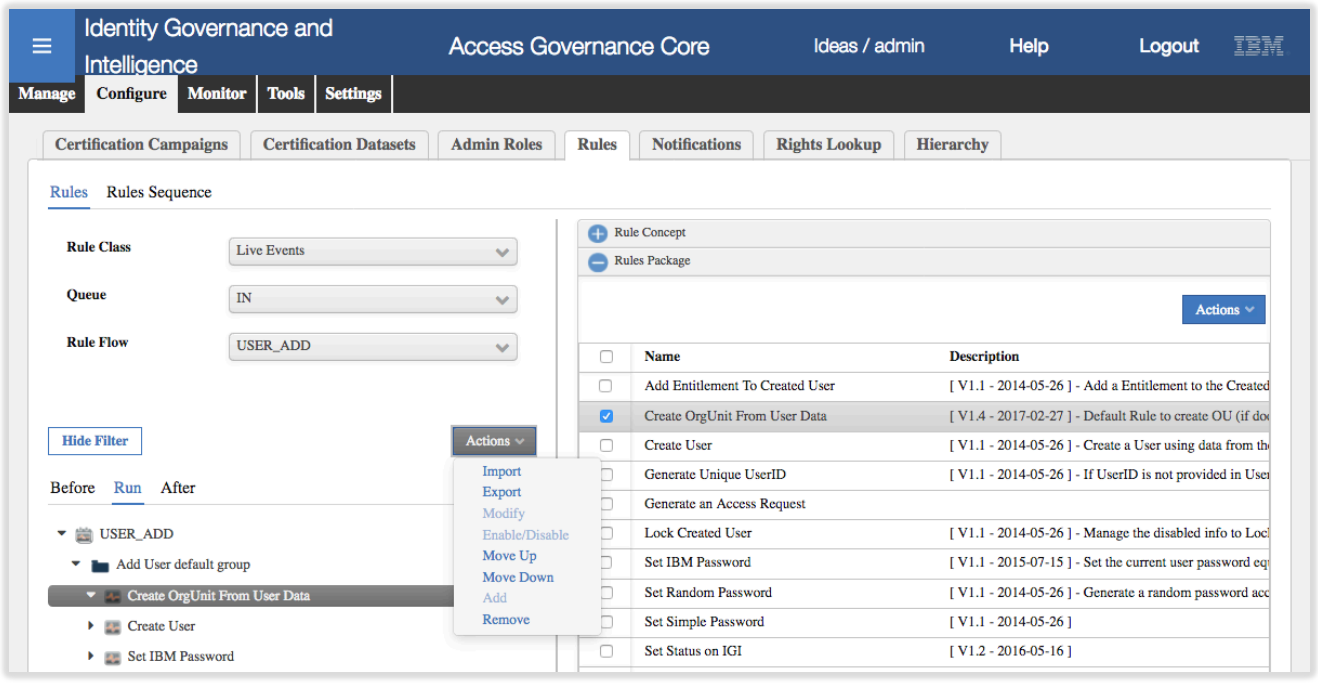
With a Rule Class, optionally a Queue, and a Rule Flow selected, the associated Rule Package is available. From this list, the rules to be executed can be added to the Rule Flow.

As mentioned above, rules can be selected from the Rule Package and added to the flow (Actions > Add in the Rule Package view). Other operations for rules and rule flows are available in the Rule Flow view.

The Actions pull-down menu in the Rule Flow view are:

- Import – For importing the XML representation of a Rule or of a Rule Sequence (Rule Flow)
- Export – For exporting a Rule or a Rule Sequence (Rule Flow)
- Modify – Modify a Rule Group (for Groups, function that is maintained for legacy reason)
- Enable/Disable – Enable or disable a Rule execution into a sequence (this option seems to be disabled in 5.2.3)
- Move Up – move a rule up the execution sequence
- Move Down – move a rule down the execution sequence
- Add – Adds a Group of Rules (function that is maintained for legacy reason)
- Remove – Removes a Rule or a Group of rules (for Groups, function that is maintained for legacy reason)

Rule Groups, shown with a folder icon, are no longer used, but some functions remain for legacy reasons.



2.4 Process for Adding a New Rule

The process for adding a new rule is:

1. Perform analysis – Define the business logic you want to implement, what operation it needs to be associated with in IGI, the objects and methods involved and the logic you need to code
2. Define/Identify a Rules Sequence (optional) – If you are creating a non-event rule you need to check in the list of Rules Sequences to see if there is a Rules Sequence to match what you need. If not, create one.
3. Select the IGI operation to associate the rule with; in the Rules view select the Rule Class, optionally a Queue (for event-based rules) and Rule Flow (either the specific event for event-based or a Rules Sequence for a non-event rule)
4. Open the Rules Package view and use **Actions > Add** to open the Rules Editor
5. Create/insert your new rule (optionally using the helper functions in the right pane) and save it
6. Check all the relevant classes are imported in the Package Import view
7. Go back to the Rules Package and select your new rule, and use **Actions > Verify** to check for errors
8. Add the new rule to the Rule Flow – select the new rule in the Rules Package and use Actions > Add to add it to the Rule Flow
9. In the Rule Flow view use **Actions > Move Up** or **Actions > Move Down** to order the new rule amongst any existing rules
10. If it is a non-event rule, you will need to go into the relevant function (e.g. campaign, workflow process/activity, account configuration) and attach/enable the rule
11. Test the rule

Note, the rules that are delivered with the product and made available after the installation, must be kept "as is", without any modification in the Rules Packages. Do not modify the product rules. If you need to customize a product rule, copy it, rename it, and modify the renamed copy. If you modify a product rule, the system might display an unpredictable behaviour. However, depending on the rule, you may add/remove it from the relevant rule flow (in some cases there must be at least one rule performing a basic function, like adding a user, whether it's a supplied rule or a custom one).

2.5 Eclipse Development Environment

Details coming.

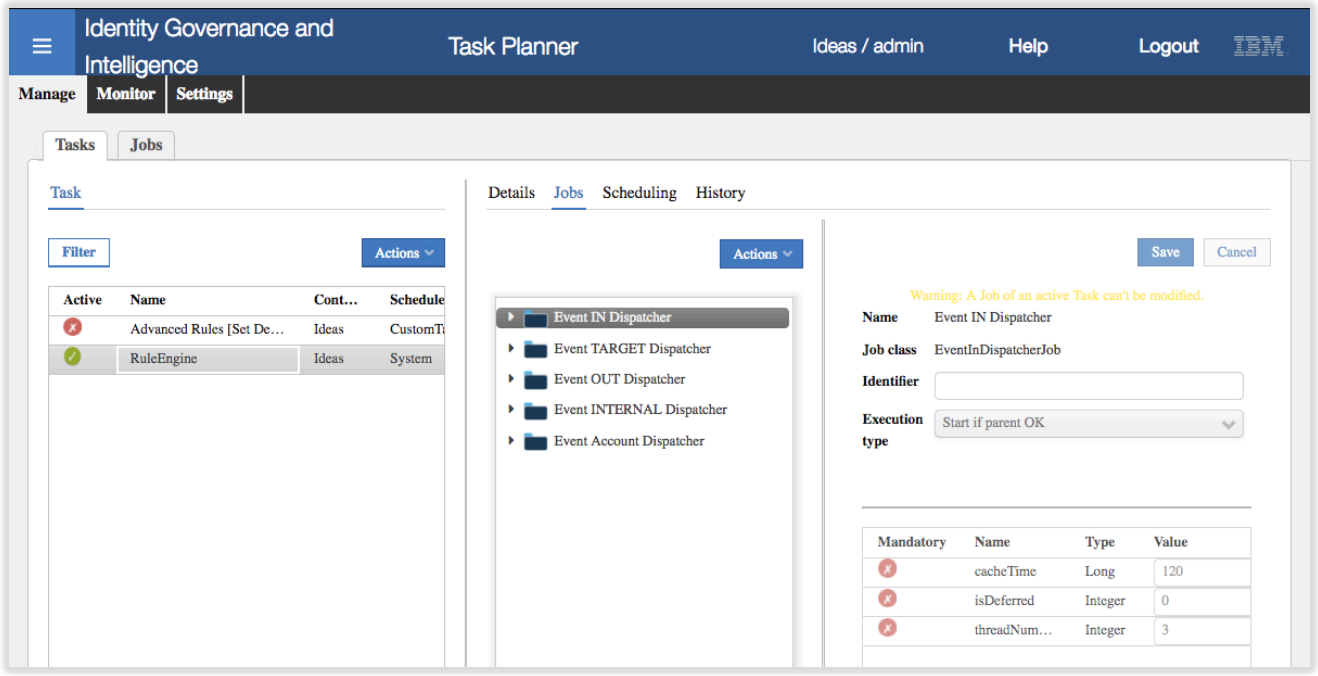
2.6 Testing Rules

Testing of rules will depend on the rule function being implemented. We will discuss testing approaches with the detailed exploration of the different rule types in later sections of the document.

2.7 The Rules Cache

Rules are interpreted/compiled when they are needed. To improve performance, a Rules Cache was introduced for event-based rules. Whilst this improves performance, it means any new rules will not get loaded until the currently cached version expires. The default cache expiry is 120 minutes (2 hours). When testing rules, you may want to reduce this cache time or disable it completely.

There is a Task, RuleEngine, that runs dispatcher jobs on each of the four queues (IN, OUT, TARGET and INTERNAL). Each of these jobs has a cacheTime parameter that is set to 120 minutes by default.



To change or disable the cache time, stop the Task, apply the change to the appropriate queue dispatcher (and Save), and restart the Task. You should only do this for the duration of the testing/debugging; once the code is ok you should revert to a longer cache time.

2.8 Error Handling, Debugging, Logging and Log Files

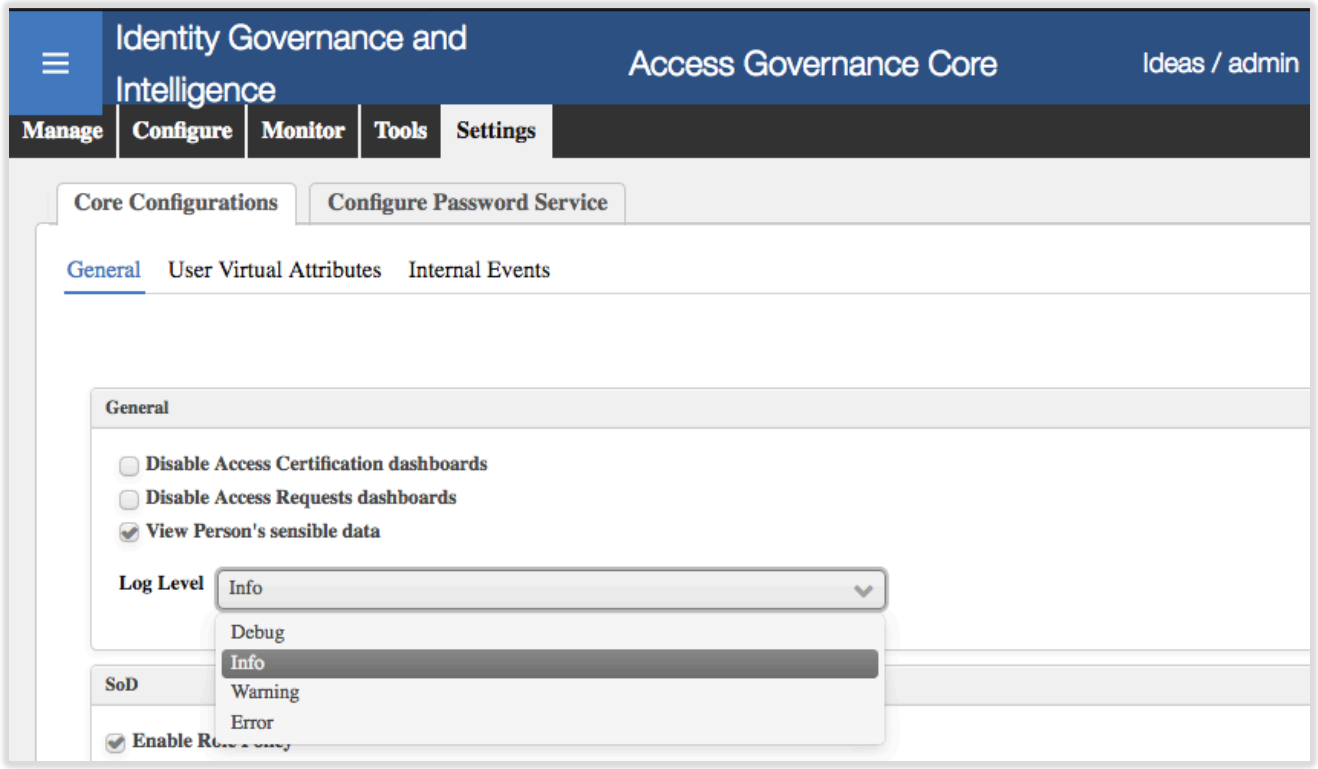
Debugging will normally involve the use of the system logging mechanism and coding logging into the rules.

2.8.1 Error Handling in Rules

The rules use the normal Java conventions for error handling, such as the try/catch blocks and the `throw new Exception("blah!")` method.

2.8.2 IGI System Logging

IGI has a system-wide logging level setting, found in **Access Governance Core > Settings > Core Configurations > General**, called Log Level (as shown below).



Four different log levels can be set:

- **Error**: Records errors only.
- **Warning**: Records errors and indications of possible errors.
- **Info**: Records errors, warnings, and information messages.
- **Debug**: Records errors, warnings, information messages, and messages associated to the debugging phase.

Thus, Debug is the most verbose and will result in logs filling a lot faster. The default setting is Info.

2.8.3 System Log Files

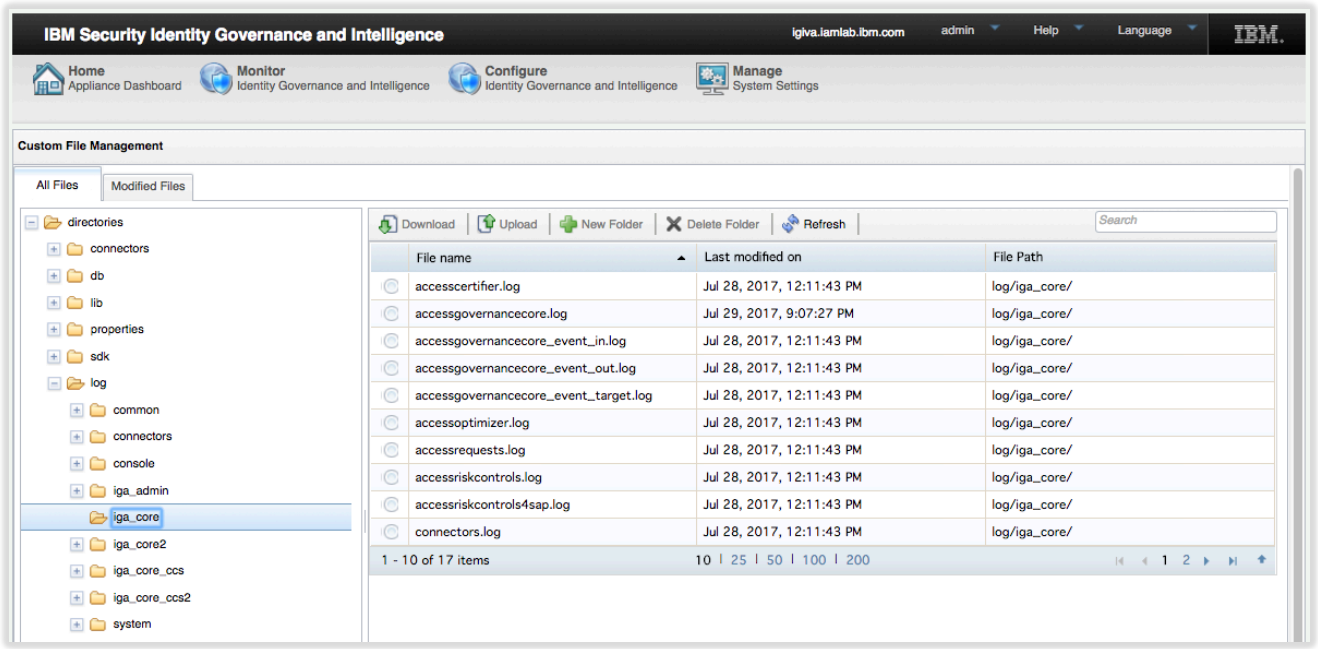
Due to the original loosely-coupled muti-tenancy architecture of Ideas, and the addition of the Identity Brokerage component and adapter, there are many logs used by the product. They fall into four categories:

- The IGI application logs,
- The Identity Brokerage logs,
- The adapter logs, and
- The adapter agent logs

As rules run within the IGI application, you will not see rules logging in any of the latter three sets of logs. However if you are developing rules that work on events flowing into or out of IGI, you may see associated behaviour in these logs. We don't cover them in detail in this document, but there is ample product documentation and support material covering these logs.

There are also system logs for the WebSphere Liberty processes and the datastores that don't directly relate to rules coding and debugging.

The IGI application logs are held on the IGI Virtual Appliance and accessed via the Local Management Interface (LMI). They are accessed via the **Configure > Custom File Management** option and live in folders under the `directories > logs` path.



The following logs are used by the rules:

Class	Logs
Live	logs\iga_core\accessgovernancecore_event_*.log
Authorization Digest	*Standard logger statements don't work in these rules.* *Instead use System.out.println() that write to IGI application server message log file.*
Account	logs\iga_core\accessgovernancecore.log
Attestation	logs\iga_core\scheduler_job.log
Hierarchy	logs\iga_core\scheduler_job.log
Advanced	logs\iga_core\scheduler_job.log
Advanced (ARCS)	logs\iga_core\accessriskcontrols4sap.log

If the `System.out.println()` method is used, messages will be written out to the IGI application server message log file.

2.8.4 Logging Methods for Rules

There are two main logging methods; `logger.debug()` and `logger.info()`. If you use logger.debug. the messages will only appear if a system log setting of Debug is used.

For example, If I include the following in an OUT event rule

```
logger.debug("$$$$ DEBUG *** Sync level: " + eventOut.getCodiceOperazione());
logger.info("$$$$ INFO *** Sync level: " + eventOut.getCodiceOperazione());
```

Then the accessgovernancecore_event_out.log will look similar to:

```
Jan 26, 2017, 9:21:36 AM DEBUG AGC:? - $$$$ DEBUG *** Sync level:
PM_1866277246650801262_admin
Jan 26, 2017, 9:21:36 AM INFO  AGC:? - $$$$ INFO *** Sync level:
PM_1866277246650801262_admin
```

There are some examples of these methods in the following section.

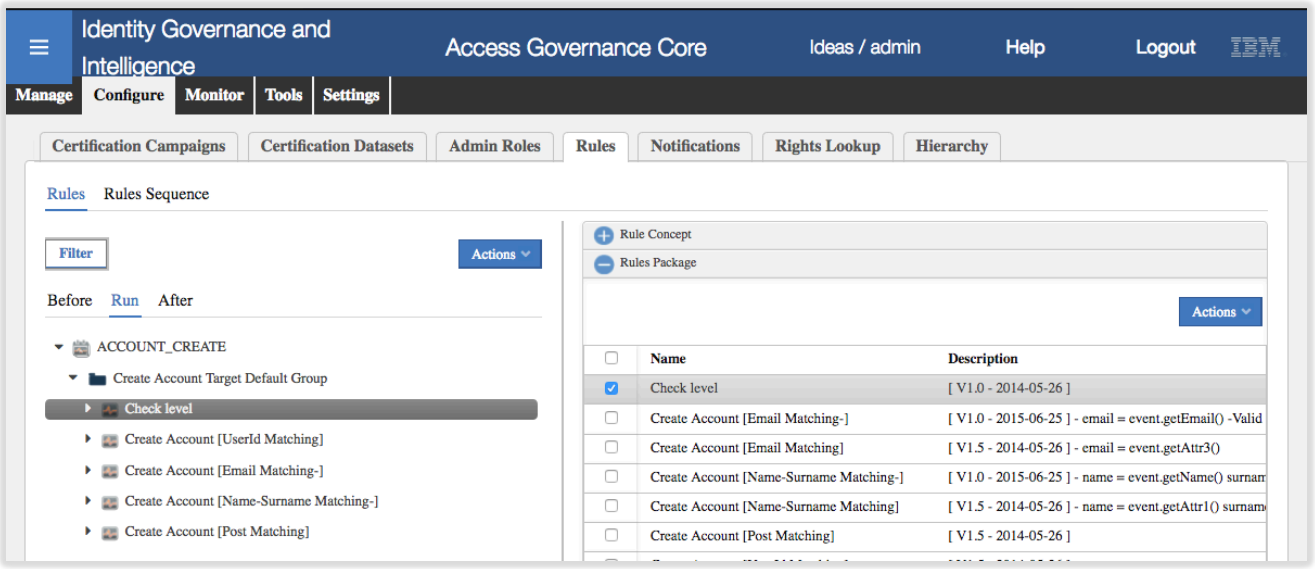
As mentioned earlier, you can also use `System.out.println()` which will write to the IGI application server (WebSphere Liberty) message log.

2.8.5 Example: Check Level Rules

For each "Live Events" Rule Flow, the EventBean is always present among the objects in the working memory. There are four kinds of EventBean, each relating to a specific queue: EventInBean, EventTargetBean, EventOutBean, EventInternalBean

The EventTargetBeans have a property named level representing the last error code. The level property is an int property and can be retrieved using the command: event.getLevel().

There is a "Check level" rule in many of the target events flows, that will check the error code prior to executing the rule flow. It only appears used in the account and permission events in the TARGET queue. For example, the Check level rule in the Live Events/TARGET/ACCOUNT_CREATE flow:



This rule has the following code:

```
when
    event : EventTargetBean(  )
then
// [ V1.0 - 2014-05-26 ]

  int syncLevel = event.getLevel();

  logger.debug("Sync level: " + syncLevel);

  if (syncLevel == EventTargetBean.USER_FOUND) {
     logger.debug("Account Already exists");
  }
```

This code is retrieving the last return code for this operation, in this case an ACCOUNT_CREATE, and using it for writing out some debug messages (using the `logger.debug()` method).

This next example is from the Live Events/TARGET/PERMISSION_ADD rule flow:

```
when
    event : EventTargetBean(  )
then
// [ V1.0 - 2014-05-26 ]

  int syncLevel = event.getLevel();

  logger.debug("Sync level: " + syncLevel);

  if (syncLevel == EventTargetBean.USER_NOT_FOUND) {
    logger.debug("Account not found");
    throw new Exception("Account " + event.getCode() + " does not exist");
  }

  if (syncLevel == EventTargetBean.PROFILE_NOT_FOUND) {
    // The following rules create the permission if it doesn't exist
    // throw new Exception("Profile not found ");
    logger.debug("Profile not found");
  }

  if (syncLevel == EventTargetBean.USER_PROFILE_FOUND) {
    logger.debug("The user already has the added profile");
  }
```

The values available are:

- USER_NOT_FOUND – Account does not exist
- USER_FOUND – Account exists
- OU_USER_NOT_FOUND – OU for user not found
- PROFILE_NOT_FOUND – Permission does not exist
- JR_PROFILE_NOT_FOUND – TBC!
- OU_PROFILE_NOT_FOUND – Permission not attached to OU
- USER_PROFILE_NOT_FOUND – No user to permission mapping
- USER_PROFILE_FOUND – A matching user to permission mapping was found

These do not appear in the other EventXXBeans, only the EventTargetBean.

2.9 Hints and Common Practices

This section provides some hints and common practices for rules in IGI.

2.9.1 Passing Data Between Rules Using the ContainerBean

If you have a series of rules in a flow, you can pass data between them using the ContainerBean. The ContainerBean is an optional object available in the working memory. It is an HashMap <String, Object>. As a HashMap it has «put» and «get» methods.

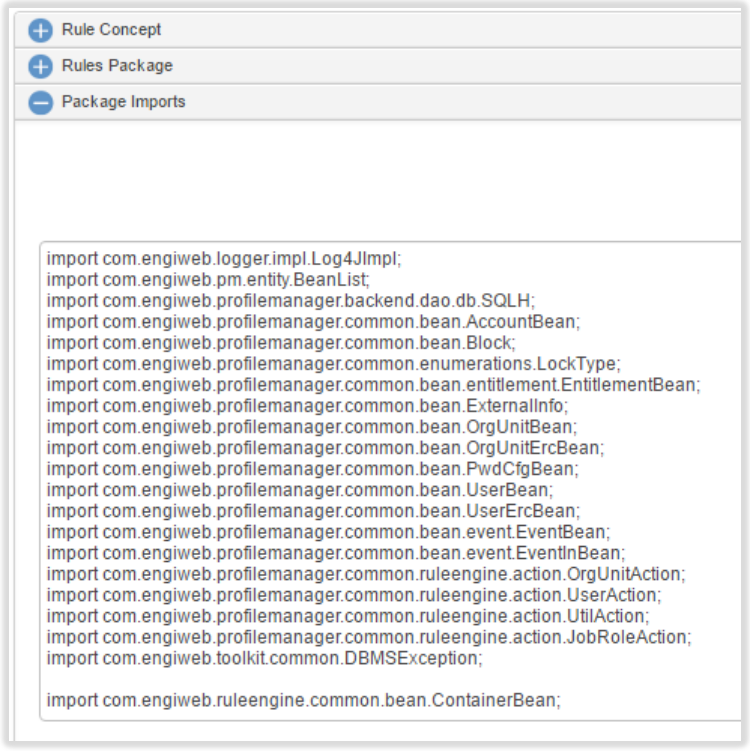
Thus you could set it up in the first rule and then access it in subsequent rules. For example, the first rule could have:

```
when
    user : UserBean(  )
    orgUnit : OrgUnitBean(  )
    extInfo : ExternalInfo(  )
    containerBean : ContainerBean(  )

then
// [ V1.1 - 2014-05-26 ]

// ContainerBean<String, Object> containerBean = new ContainerBean<String, Object>();

YourObject yourObject = new YourObject();
containerBean.put("yourKey", yourObject);
...
```

Then subsequent rules could have:

```
when
    user : UserBean(  )
    orgUnit : OrgUnitBean(  )
    extInfo : ExternalInfo(  )
    containerBean : ContainerBean(  )

then
// [ V1.1 - 2014-05-26 ]

// ContainerBean<String, Object> containerBean = new ContainerBean<String, Object>();

YourObject yourObject = containerBean.get("yourKey");
...
```

You must include the container bean import command in the Package Imports section for any flow using it.



2.9.2 Use of 1L for Ideas Account Configuration and OU root

The first system objects defined normally have an identifier of 1 Long (“1L”). This is often used as a shortcut in rules.

The Ideas account for every user is always the first account allocated and it has an identifier of 1L. It is common to see code like

```
aBean.setPwdcfg_id(1L);
```

Which is referring to the account configuration (poorly named “Pwdcfg”) of the Ideas account. In this case it’s setting aBean to the Ideas account config.

Similarly the id of 1L when referring to an org unit will always mean the root of the OU tree.

You will find `1L` and `0L` used throughout for True and False. For example setting an event state to 1L sets it to True (`event.setState(1l);`) to prevent it from being deleted.

2.9.3 Understanding the Operation Code (codOperation)

All operations will have an operation code (“CodiceOperazione” or codOperation), that may be used in rules. It is visible in the OUT queue in the UI (AGC > Monitor > OUT Queue).

The codOperation is a good mechanism to understand what function or module generated events.

Generally it is composed of an initial tag plus a random number (although you will find codes that are strings of words). The number could be a timestamp or a random number. The tag is an identifier of component which generated events.

For example if you assign an Entitlement to a user using AGC UI you will see many 'Add Entitlement' events in the out queue (one for each permission in the entitlement) with same code operation in a format like PM_<rnd>.

A tag of “PM” (short for Profile Manager) identifies the AGC console or a client that called by an EJB without providing its own codOperation. This means any external client that interacts with IGI through EJB or Rest can provide its own codOperation key, and that can be used to trace which operation has been done, and optionally apply different actions.

When an action comes from a Target queue event, the generated codOperation is MR_TARGET_<number>, where number is the ID of event in the Target queue. When an action comes from an IN queue event, codOperation is MR_IN_number where number is ID of eventIN queue. There most useful place for analyzing and determining actions on the codOperation is in the OUT queue events.

Some rules, particularly older ones will check the operation code (e.g. extract the reuqest Id from an access request).

3 Rules and Queues for Enhancing Data Flows

One of the main uses of rules is to enhance or extend the processing of data flowing into or our of IGI. This chapter looks at the different flows and how rules can be applied.

3.1 Introduction to Event-Based Rules

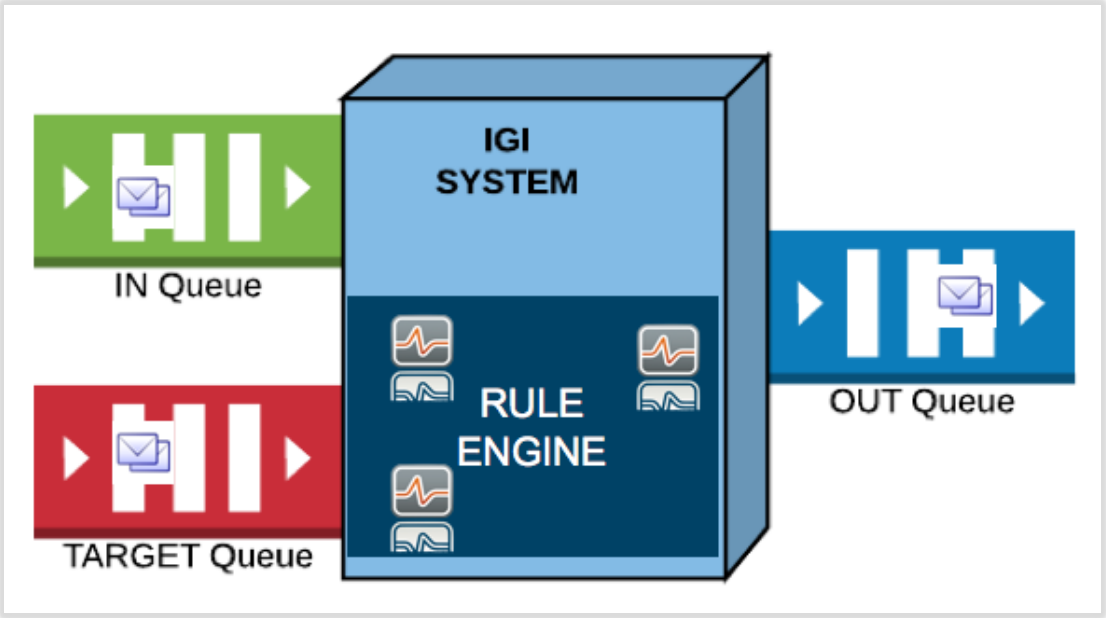
Event-based rules operate on events flowing through queues. Related to this are the pre- and post-mapping rules used by the Enterprise Connector framework.

3.1.1 Rules, Queues and Data Flows

The primary interface for external systems working with IGI is Queues. External systems, like a HR Feed mechanism or an Enterprise Connector or Brokerage Adapter will write events to an in queue to be processed by IGI. IGI will write out events to out queues to be consumed by external systems, like connectors and adapters. Events are messages in the queues.

Rules can be applied to events flowing through the queues; rules are processes that consume events.

The relationship between queues, events, the rule engine and rules is shown in the following figure.



For data flowing into IGI, the events are written to the queues, processed by rules and applied to IGI. For data flowing out of IGI, events are processed by rules, written to the out queues and consumed by applications.

There are four main queues used in this mechanism the operate on different data types and operations.

Queue	Objects	Use
IN	People, OUs	Incoming events from a repository of people (users) and OUs (organizational structure) via some external HR Feed mechanism
TARGET	Accounts, Permissions	Incoming accounts and access rights (permissions) from an account repository via an Enterprise Connector or Brokerage Adapter, or other mechanism. This often referred to as "reconciliation"
OUT	Accounts, Permissions	Outgoing changes to accounts and access rights (permissions) from IGI via an Enterprise Connector, Brokerage Adapter or other mechanism. This is often referred to as "provisioning"
INTERNAL	Applications OUs, Users Entitlements	Internal processing of events, when enabled, allows rules to be triggered on some events for these objects.

There is a new queue for processing outgoing user events, but this is restricted for use with the ISIM integration (ISIGADI).

The processing of events on these queues is covered in detail below.

3.1.2 The Rule Engine and Before Rules

All events have an Operation field that identifies what the event represents, like accountCreate, addPermission, etc. What we call the Rule Engine is only a job that checks the queues with a defined frequency (by default each 20 seconds) to see if there are some rows with status = 0 (meaning 'unprocessed'). When events are found, it processes them starting from the oldest one.

The Rule Engine gets rows as is from queue and calls the Before rule. At this time in the working memory there is only the EventBean Object (TargetEventBean or OutEventBean ... based on queue). The Before rule can be used to 'store' all common proceesing rather than repeating the code across each rule.

It can also be used to modify the event itself. You can change operation code, or change permission name etc. Then, the Rule Engine will look at the event and based on operation value calls the dedicated Event flow.

Of course the Rule Engine performs some searches on the DB to take the most important objects to put in the working memory and make them available to the rule flow.

3.1.3 Mapping Rules for the Enterprise Connector Framework

This section talks about rules related to the events flowing into and out of IGI. One of the main generator/consumer of events to/from the IGI queues are the legacy Enterprise Connectors and Identity Brokerage Adapters, both which use the Enterprise Connector framework to communicate with the IN, OUT and Target queues. This framework also leverages rules for data transformation and other functions.

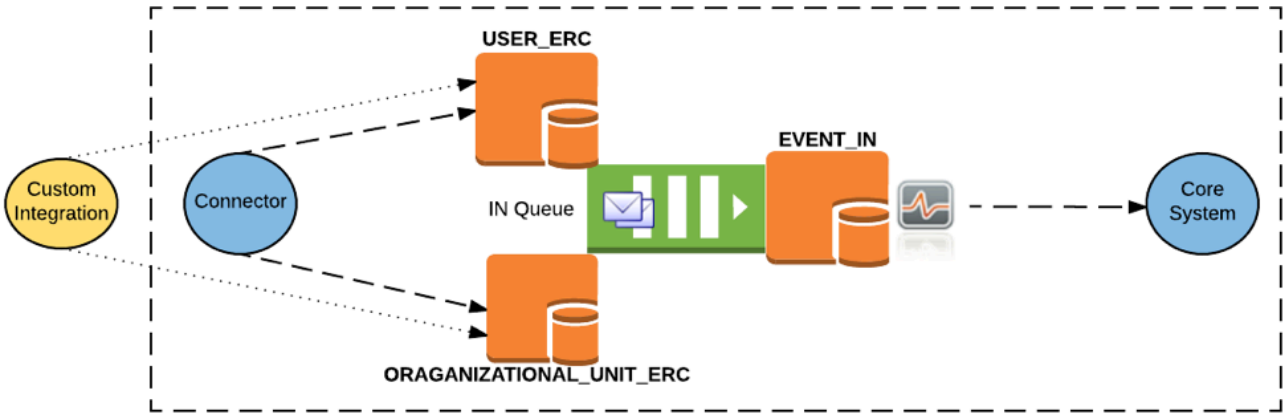
Any analysis of custom logic to implement business use cases, should consider the Enterprise Connector rules along with the Event-based rules. We will look at the Enterprise Connector flows and rules in the next chapter (Data Mapping Rules for Enterprise Connector Flows on page 70).

3.2 Processing IN User and OU Events

The IN queue is used to process incoming user and OU events. This section looks at the flows, events and rules that can be applied.

3.2.1 Event Flows and Events for the IN Queue

The IN event flow is shown in the following figure.



External systems, like a custom HR Feed mechanism or one of the identity adapters (SAP or PeopleSoft), will write user and OU events into the USER_ERC (for user) and ORGANIZATIONAL_UNIT_ERC (for OU) tables. Internal processing will combine these events into the one table, EVENT_IN, which is the IN queue.

Rules can operate on the events in the IN queue. They are normally processed as Live Events.

Input Events	Rule Class	Queue	Rule Flow	Notes
Create User	Live Events	IN	USER_ADD	
	Live Events	IN	USER_BEFORE	1.
Modify User	Live Events	IN	USER_MODIFY	
Remove User	Live Events	IN	USER_REMOVE	
Move User	Live Events	IN	USER_MOVE	2.
Create OU	Live Events	IN	ORGUNIT_ADD	
	Live Events	IN	ORGUNIT_BEFORE	1.
Modify OU	Live Events	IN	ORGUNIT_MODIFY	
Rename OU	Live Events	IN	ORGUNIT_REMOVE	

Notes:

1. The ***_BEFORE Rule Flows provide a mechanism to have some common code used by all of the other flows. For example, you may have some standard code for processing users that you don't want to replicate into the USER_ADD, USER_MODIFY and USER_REMOVE flows, so you can code it into the USER_BEFORE flow.
2. This is a rule flow for moving a user from one OU to another

There is also a special category of Deferred Events, where rules can be be run deferred and scheduled through the Task Planner. The rules have a different class but operate on the same IN queue and have the same Rule Flows.

Input Events	Rule Class	Queue	Rule Flow	Notes
Create User (Deferred)	Deferred	IN	USER_ADD	
	Deferred	IN	USER_BEFORE	1.
Modify User (Deferred)	Deferred	IN	USER_MODIFY	
Remove User (Deferred)	Deferred	IN	USER_REMOVE	
Move User (Deferred)	Deferred	IN	USER_MOVE	2.
Create OU (Deferred)	Deferred	IN	ORGUNIT_ADD	
	Deferred	IN	ORGUNIT_BEFORE	1.
Modify OU (Deferred)	Deferred	IN	ORGUNIT_MODIFY	
Rename OU (Deferred)	Deferred	IN	ORGUNIT_REMOVE	

The OU deferred events are not currently (IGI 5.2.3) enabled.

3.2.2 Writing Rules on IN User / OU Events

The rules written here will be triggering off a changed (new, modified, moved or deleted) person or org unit in the HR system. If there was merely a need to modify the data coming into IGI, you could use the mapping rules in the Enterprise Connector framework (or something in your custom HR Feed mechanism).

Event-based rules operating on the IN queue tend to drive other associated activity in IGI. For example:

- A new OU triggers the assignment of an administrator to manage it
- A new person will need to be placed in an OU, but if one doesn't exist it may need to be created
- A unique master userid needs to be generated
- Automatically grant a specific account/access where you don't want to use a hierarchy

There are rules that must be present, either using the supplied rules or custom rules, to add a user (e.g. `UserAction.add(sql, userBean, orgUnitBean, externalInfo);`) and add an org unit (e.g. `OrgUnitAction.add(sql, orgUnit, false, false);`). If you remove the supplied rules and do not add your own to replicate these functions, you will not get users/OUs created. There may be similar default rules for modify and delete events. You should check the default rules before removing or changing them.

3.2.3 Objects Available for IN Events

The following beans may be available in the working memory (based on the event):

- *EventBean() / EventInBean()* – common event details (like operation type)
- *UserBean()* – the User as it is, or will be, in IGI
- *ExternalInfo()* – the additional attributes for user objects
- *UserErcBean()* – the incoming User attributes
- *OrgUnitBean()* – the Org Unit as it is, or will be, in IGI
- *OrgUnitErcBean()* – the incoming Org Unit attributes

These can be found in the JavaDoc but aren't very well defined.

The *EventBean()* (which has a subclass of EventInBean()) includes an operation attribute (getOperation()) that is a numeric representation of the operation the event represents, including:

- 1 = ADD User
- 2 = MODIFY User
- 3 = REMOVE User

- 9 = ADD OU
- 10 = MODIFY OU
- 11 = REMOVE OU
- 12 = MOVE User

The complete list can be found at https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/db_tables/ref/db_EVENT_IN.html

The rules can also access other objects in IGI, such as roles/permissions and accounts, as shown in the following examples.

3.2.4 Example: Set Random Password for Ideas Account on New User

When any new person is defined in IGI, they are automatically granted an Ideas account (their account to access the IGI Service Center). The following code will generate a random password and assign it to the Ideas account.

It demonstrates some of the UserActions as well as working with the Ideas account for a user.

```
when
    user : UserBean(  )
    orgUnit : OrgUnitBean(  )
    extInfo : ExternalInfo(  )
then
// [ V1.1 - 2014-05-26 ]

  // Gets account cfg
  PwdCfgBean ideasAccountCfg = new PwdCfgBean();
  ideasAccountCfg.setId(1L);

  String pwd = UserAction.getRandomPwd(sql, ideasAccountCfg);

  AccountBean userAccount = new AccountBean();
  userAccount.setPwdcfg_id(ideasAccountCfg.getId());

  BeanList accounts = UserAction.findAccount(sql, user, userAccount);
  userAccount = (AccountBean) accounts.get(0);

  UserAction.changePwd(sql, userAccount, pwd);

  logger.debug("Random password set");
```

The when clause is assigning the UserBean, OrgUnitBean and ExternalInfo beans to variables (if any are null, the rule won't fire). It does not use either the OrgUnitBean of ExternalInfo beans.

The rule will:

- First get the account configuration (PwdCfgBean) for the Ideas account. Note the use of the `setId(1L)` which is shorthand for the Ideas account (it is always the first defined, this Id = 1).
- Next it will generate a random password (`UserAction.getRandomPwd()`) using the Ideas account configuration.
- Then it will create a new AccountBean object and set the ID to the Ideas Id and then search for an account for this user matching that config (`UserAction.findAccount()`)

- Finally it will set the password of the ideas account to the randomly generated one (`UserAction.changePwd()`).

This is a fairly simple rule. It would rely on some email notification mechanism to inform the new user of their IGI password.

3.2.5 Example: Set New User as Department Manager if A Manager

Every new user added to IGI is assigned to an Org Unit, which may represent a department. Some organisations may define a department manager admin role for access request approval or other managerial duties.

In this example, the new user is checked to see if they are flagged as a Manager (from HR) and if so, assigns them to the Department Manager admin role with a scope of the OU that they are assigned to.

```
when
    event : EventInBean(  )
    userErcBean : UserErcBean(  )
    userBean : UserBean(  )
    extInfoBean : ExternalInfo(  )
    orgUnitBean : OrgUnitBean(  )
then
// [ V1.1 - 2014-05-26 ]

  String MANAGER_ROLE_NAME = "Department Manager";
  String IS_MANAGER_ATTR = "ATTR2"; // Y/N

  String isManager = (String) userErcBean.getAttribute(IS_MANAGER_ATTR);

  if (isManager != null) {

    if (isManager.toUpperCase().equals("Y")) {

      // Get manager role object
      EntitlementBean role = UtilAction.findEntitlementByName(sql, MANAGER_ROLE_NAME);
      if (role == null) {
        throw new Exception("Role with name " + MANAGER_ROLE_NAME + " not found!");
      }

      BeanList roles = new BeanList();
      roles.add(role);

      OrgUnitAction.addRoles(sql, orgUnitBean, roles, false);
      UserAction.addRole(sql, userBean, orgUnitBean, roles, null, null, false, false);
      UtilAction.addResourcesToEmployment(sql, userBean, role, orgUnitBean);
    }
  }
```

The when clause is assigning the EventInBean, UserErcBean, UserBean, OrgUnitBean and ExternalInfo beans to variables (if any are null, the rule won't fire). It does not use the ExternalInfo bean.

The first part of the rule sets two hardcoded variables; the name of the manager role ("Department Manager") and the attribute holding the IS_MANAGER flag ("ATTR2"). These values are obviously deployment-specific.

The rule flow is:

- It retrieves the IS_MANAGER value from the userErcBean, which is held in ATTR2 (`userErcBean.getAttributes(IS_MANAGER_ATTR);`)
- If the IS_MANAGER flag is set and equal to (uppercase) "Y") it will:
    - Get (build) the bean for the "Department Manager" admin role (`UtilAction.findEntitlementByName();`)
    - Build a new BeanList (array) and add the found "Department Manager" admin role bean to it
    - Add the "Department Manager" admin role to the OU to set the visibility scope (I assume it will ignore if it's already assigned) with the `OrgUnitAction.addRoles()` method.
    - Add the user to the admin role (`UserAction.addRole();`)
    - Set the admin role scope as this org unit for this user in the role (`UtilAction.addResourcesToEmployment();`)

This is a good example of using the user and org unit to go find and set other objects in IGI using different actions.

3.2.6 Example: User Move Triggers Continuous Certification Campaign

IGI has the concept of continuous certification campaigns. Rather than running a campaign for a fixed period of time on a static dataset, a campaign can be fed changes continuously that a reviewer must review.

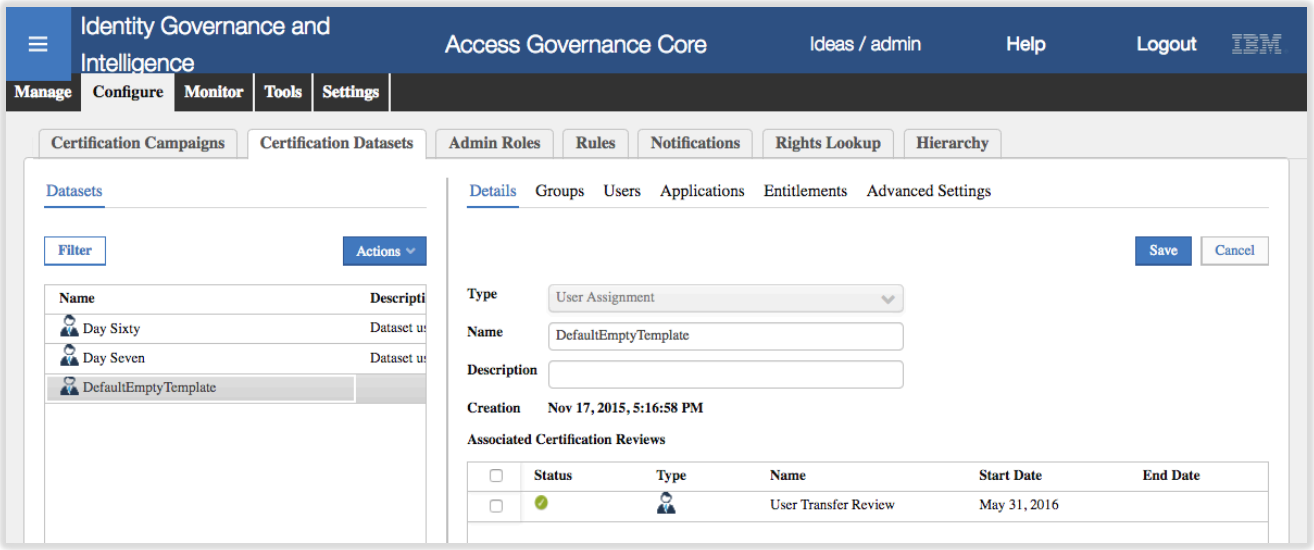
In this example a user move event (i.e. the user has changed OU) will put the entitlements for that user into the certification dataset for a continuous campaign.

```
when
    userBean : UserBean(  )
    orgUnitBean : OrgUnitBean(  )
then
// [ V1.3 - 2015-04-22 ]

  // Templatename --> DefaultEmptyTemplate included in User Transfer Campaign

  String tName = "DefaultEmptyTemplate";

  TemplateBean templateBean = new TemplateBean();
  templateBean.setName(tName);

  TemplateDAO templateDAO = new TemplateDAO(logger);
  templateDAO.setDAO(sql);

  BeanList blTemplateBean = templateDAO.find(templateBean, new Paging(4));

  if (blTemplateBean.size()==0) {
    throw new Exception("Template does not exists!");
  }
  templateBean = (TemplateBean) blTemplateBean.get(0);

  BeanList entitlements = UserAction.findJobRoles(sql, userBean);

  BeanList listBean = new BeanList();
  for (int i = 0; i < entitlements.size(); i++) {
    AbstractBean[] element = new AbstractBean[2];
    element[0] = userBean;
    element[1] = (EntitlementBean) entitlements.get(i);
    listBean.add(element);
  }

  if (listBean.size() > 0) {
```

```
    templateDAO.addEntity(listBean, AttestationRes.TEMPLATE_ENTITY_USERENT, templateBean,
AttestationTypes.PERSON_ENTITLEMENT.getValue());
  }
```

The when clause is referencing both the userBean and the orgUnitBean although the code doesn't seem to do anything with the orgUnitBean.

The term "template" refers to a Certification Dataset (old product terminology?). The name of the certification dataset ("DefaultEmptyTemplate") is hardcoded into a string. As per the comments, this dataset is assoicated with the "User Transfer Campaign" continuous certification campaign.



The code:

- Creates a new TemplateBean and sets the name to the certification dataset name
- Creates a new TemplateDAO (data access object)
- It uses the data access object to search through the certification datasets looking for one matching the new TemplateBean (i.e. our "DefaultEmptyTemplate"), this is the `templateDAO.find()` method
- If found, it grabs the first element (there should only be one, the "DefaultEmptyTemplate")
- It looks for all entitlements associated with the user (it is not clear whether it only gets internal roles (IT & Business Roles) or all entitlements, I assume it gets all)
- It creates a new BeanList (array) for the entitlements found and for each entitlement, sets two values for the corresponding BeanList entry (row in the array) – the userBean and the entitlementBean for the entitlement
- Finally, if there were entitlements found for the user, it adds each one to the certification dataset (using `templateDAO.addEntity();`). The AttestationRes.TEMPLATE_ENTITY_USERENT and AttestationTypes.PERSON_ENTITLEMENT is telling the DAO that both the dataset and campaign are "User Entitlement" types.

The continuous certification campaign mechanism is not clearly defined, so this code example gives a good understanding of how you can dynamically populate the certification dataset using rules.

3.2.7 Example: User Move Enforcing Default Entitlements

The supplied User Move rules include the actual method to move the user from one OU to another. Hover the standard method, UtilAction.moveUser(sql, userBean, orgUnitBean);, only moves the user, it does not enforce any Default Entitlements set on the new OU.

The following rule will do this.

```
when
    userBean : UserBean(  )
    orgUnitBean : OrgUnitBean(  )
then
// Check if the new OU is actually different
if (userBean.getOrganizationalunit_code().equalsIgnoreCase(orgUnitBean.getCode())) {
  logger.debug("User " + userBean.getCode() + " not moved.");
  return;
}

// Make sure all the user entitlements will be available on the new OU
BeanList entitlements = UserAction.findJobRoles(sql, userBean);

// Set Visibility Violation
for (int i = 0; i < entitlements.size(); i++) {
  EntitlementBean tmp = (EntitlementBean) entitlements.get(i);

  BeanList entAssignedToOU = OrgUnitAction.findEntitlementByOU(sql, false, tmp, null,
orgUnitBean, null);
  if (entAssignedToOU == null || entAssignedToOU.isEmpty()) {
    tmp.setVisibilityViolation(1L);
  }
}

// Add entitlement setted in VV to OU
OrgUnitAction.addRoles(sql, orgUnitBean, entitlements, false);

// Move the user
UtilAction.moveUser(sql, userBean, orgUnitBean);
```

The first part of the code, above, is similar to the supplied rule, but has the added checks for the Visibility Violations and sets the VV flag if found.

The next section of code will build a search bean for the entitlements with the Default flag set to true.

```
// Assign default entitlements of OU
EntitlementBean entBeanDefault = new EntitlementBean();
entBeanDefault.setDefaultOption(true);
```

The last bit of code will search for all entitlements on the new OU, with the Default flag set, and for each one found, will add the entitlement to the user.

```
BeanList entsDefault = OrgUnitAction.findEntitlementByOU(sql, false, entBeanDefault, null,
orgUnitBean, null);
if (!entsDefault.isEmpty()) {
  for (int k = 0; k < entsDefault.size(); k++) {
    EntitlementBean role = (EntitlementBean) entsDefault.get(k);
    BeanList roles = JobRoleAction.find(sql, role);
    if (roles == null || roles.isEmpty()) {
      throw new Exception("Role : " + role.getName() + " not found!");
    }
    UserAction.addRole(sql, userBean, orgUnitBean, roles, null, null, false, false);
    logger.info("Added role --> " + role.getName());
```

```
    }
}

logger.debug("User " + userBean.getCode() + " moved to " + orgUnitBean.getCode());
```

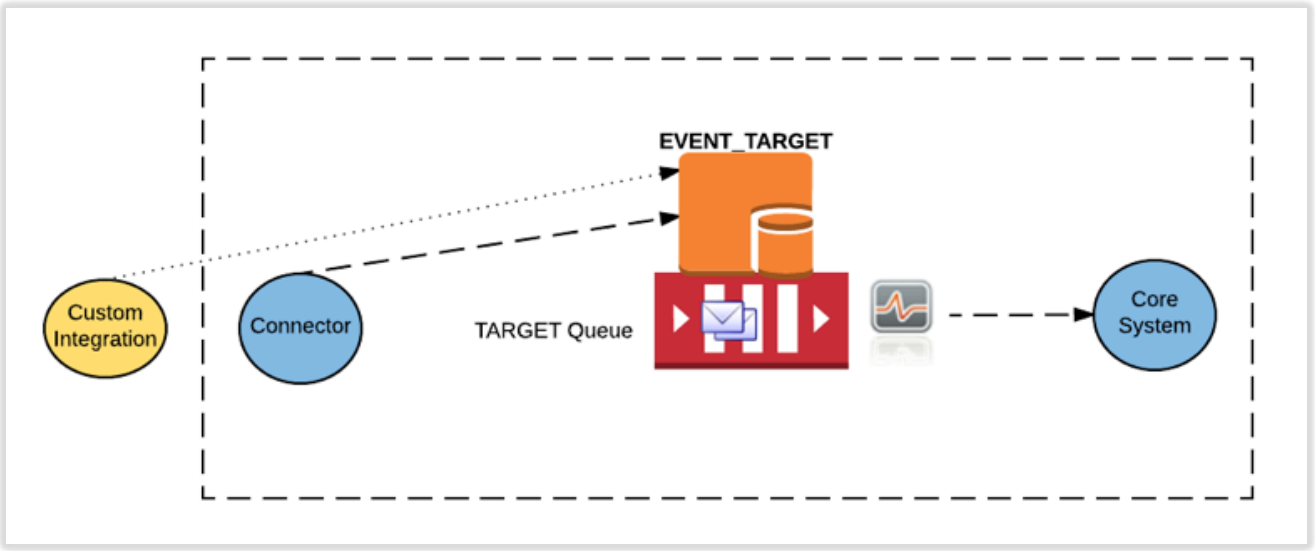
Note that this example is only adding the entitlements set as default on the new OU; it is not removing entitlements that were set as default on the old OU.

3.3 Processing TARGET Account and Permission Events

The TARGET (or TGT) queue is used to process incoming account and permission events. This section looks at the flows, events and rules that can be applied.

3.3.1 Event Flows and Events for the TARGET Queue

The TARGET event flow is shown in the following figure.



Account and permission events are written to the EVENT_TARGET table and processed as events on the TARGET queue. These events may come from the Enterprise Connector framework (either a legacy Enterprise Connector or a Brokerage Adapter) for from some other custom integration. Unlike the IN queue, there is only one database table involved holding both account and permission (sometimes called access) events.

Rules can operate on the events in the TARGET queue. They are only processed as Live Events (there are no Deferred Events for the TARGET queue).

Input Events	Rule Class	Queue	Rule Flow	Notes
Create User	Live Events	TARGET	ACCOUNT_CREATE	1.
Modify User	Live Events	TARGET	ACCOUNT_MODIFY	1.
Reset Password	Live Events	TARGET	ACCOUNT_PWDCHANGE	
Disable User	Live Events	TARGET	ACCOUNT_DISABLE	1., 2.
Enable User	Live Events	TARGET	ACCOUNT_ENABLE	1., 2.
Unmatched User (ERC only)	Live Events	TARGET	ACCOUNT_UNMATCHED	1., 3.
Remove User	Live Events	TARGET	ACCOUNT_REMOVE	1.
Add Permission	Live Events	TARGET	PERMISSION_ADD	

Remove Permission	Live Events	TARGET	PERMISSION_REMOVE	
Add Right	Live Events	TARGET	RIGHT_ADD	
Remove Right	Live Events	TARGET	RIGHT_REMOVE	
Create External Role	Live Events	TARGET	ROLE_CREATING	
Entitlement Add Child	Live Events	TARGET	ROLE_ADD_CHILD	
Entitlement Remove Child	Live Events	TARGET	ROLE_REMOVE_CHILD	
Delete External Role	Live Events	TARGET	ROLE_REMOVING	
Add External Role Child	Live Events	TARGET	<not defined>	4.
Remove External Role Child	Live Events	TARGET	<not defined>	4.
	Live Events	TARGET	BEFORE	5.

NOTES:

1. There is some confusion with the use of "user" with respect to the TARGET queue. The TARGET queue will only ever process accounts, but some systems (like SAP) call their accounts "users". Do not confuse this with users (People) in the IN queue events.
2. The Enable/Disable mechanism is supported in IGI, however not all target systems have a active/inactive flag and not all adapters support setting such a flag.
3. The Unmatched user operation is only relevant for Enterprise Connectors (ERC)?
4. These events have no associated rule flow and this cannot be extended/enhanced
5. There is a common BEFORE rule that can run before any other, event-specific, rule. This is like a common entry point and can be used to conserve rules coding/management (rather than replicating some code across multiple rules, put it in the BEFORE event flow).

Thus, the TARGET queue is processing events for four objects arriving at IGI:

- **Accounts** – processing incoming add, modify and delete events. This includes events sent from the target system to reset the password (IGI currently doesn't support a reverse password synch mechanism) and a state change (like suspend/restore). It is unlikely the Brokerage Adapters, via the Enterprise Connector framework will send disable/enable user events – any account change will come as a modify user event with attributes.

- **Permissions** – a permission, in the context of a TARGET queue event, is the assignment of an account to a target system permission.

- **Rights** – a mapping of a right on a permission for a user. For example a RACF account may be mapped to a RACF group with a default access permission (a "right" in IGI terminology) of READ.

- **Roles** – roles, or external roles, are external system access rights, which are (or will become) Permissions or External Roles in IGI. The term "external role" is used to cover both, and distinguish them from internal roles (i.e. IT Roles and Business Roles). They are equivalent to supporting data in the Identity Brokerage adapters.

For example, if a reconciliation was run against a new AD system, it would return:

- A series of "Create User" events – one for each newly-discovered AD accounts,
- A series of "Create External Role" events – one for each newly-discovered AD groups, and
- A series of "Add Permission" events – one for each AD account to AD group mapping found

3.3.2 Objects Available for TARGET Events

The following beans may be available in the working memory (based on the event):

- *EventBean()* / *EventTargetBean()* – common event details (like operation type)
- *UserBean* – user object (i.e. user for this account/permission mapping/right mapping)
- *AccountBean* – account object
- *AccountAttrValueList* – account extended attributes object, this is an object for a variable list of attributes. It has methods to add, remove, search and perform other functions on the PwdManagementAttrValBean list
- *OrgUnitBean* – OU object
- *RightsCnt* – rights informtion ?
- *SyncStateBean* – this is not docuemtned in the JavaDoc, but appears to be for permission/rights mapping
- *EntStateBean* – this is the entitlement state bean

These can be found in the JavaDoc but aren't very well defined. Not every bean will be defined for every event type, so you need to be careful if expecting beans to be available (e.g. processing an account event and expecting there to be a person associated).

The *EventBean()* (which has a subclass of EventTargetBean()) includes an operation attribute (getOperation()) that is a numeric representation of the operation the event represents, including:

- 1 = ADD Permission
- 2 = REMOVE Permission
- 10 = CREATE Account
- 11 = DELETE Account
- 12 = ADD Right
- 13 = REMOVE Right
- 25 = CREATE External Role
- 26 = REMOVE External Role

The complete list an be found at
https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/db_tables/ref/db_EVENT_TARGET.html

The rules can also access other objects in IGI, such as roles/permissions and accounts, as shown in the following examples.

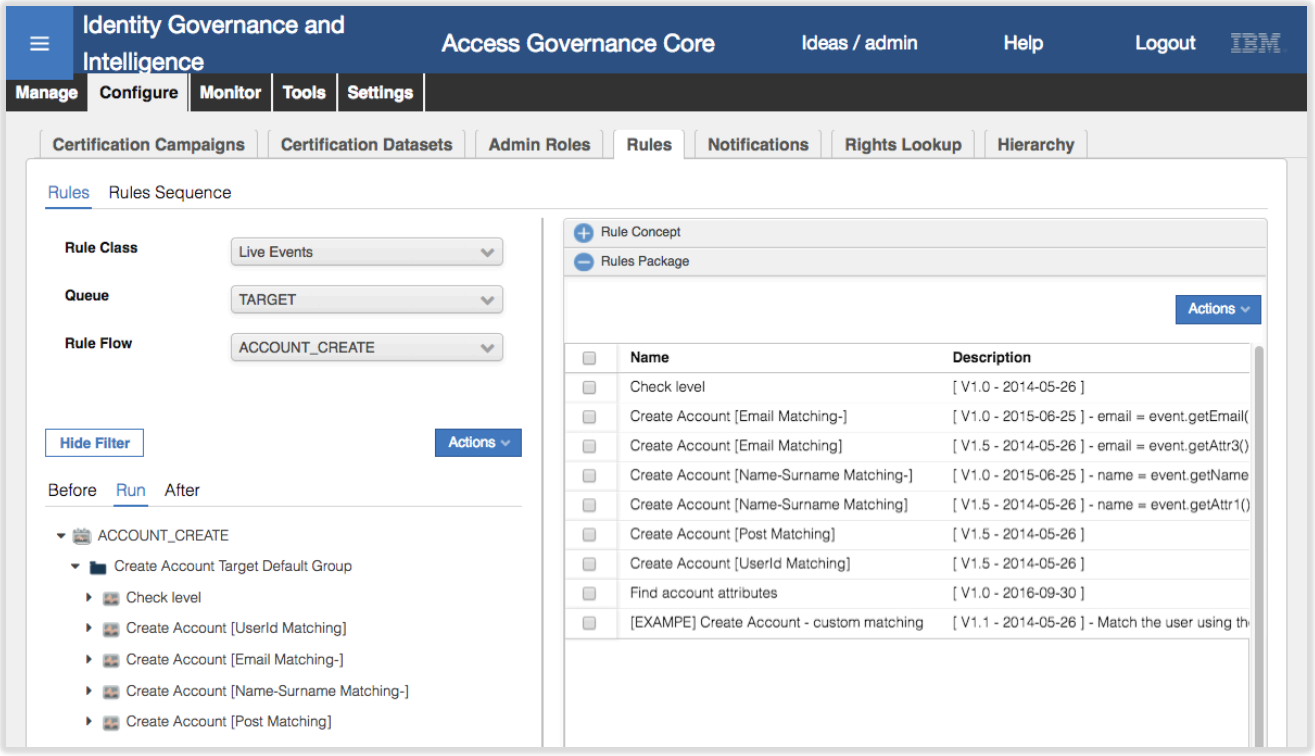
3.3.3 Example: Process a New Account from a Target

Whenever a reconciliation is run against a target system, whether it is the first or subsequent, you may get accounts sent to IGI that IGI wasn't aware of, along with any account-permission mappings. As IGI needs to associate people with permissions, it needs to match the new account with an existing user or create it in IGI as an unmatched account. There will normally be some business logic to try to find a matching user, such as looking up users by a common userid, email address or name found in the account attributes.

This example is based on the supplied code assigned to the Live Events/TARGET/ACCOUNT_CREATE event and has the following flow:

1. **Check Level** – check to see if IGI already has an account-user match for this account
2. **Create Account [Userid Matching]** – attempt to find a user by userid (account id) and create the account (or update if it's already there as an unmatched account)

3. **Create Account [Email Matching-]** – attemps to find a user by email address and create the account (or update if it's already there as an unmatched account)
4. **Create Account [Name-Surname Matching-]** – attempt to find a user by firstname and surname and create the account (or update if it's already there as an unmatched account)
5. **Create Account [Post Matching]** – if the new account has fallen through all of the matching attemps without matching, go create an unmatched account in IGI for it



Whilst there are discrete rules for the different sections, as they are in a single flow, the rules engine will process one after the other in the order shown and each needs to check for results from previous steps. Notice that there are many rules in the Rules Package, but only some of them are being used in the rule flow.

We will walk through each rule.

Check Level Rule

The code for this is:

```
when
    event : EventTargetBean(  )
then
// [ V1.0 - 2014-05-26 ]

  int syncLevel = event.getLevel();

  logger.debug("Sync level: " + syncLevel);

  if (syncLevel == EventTargetBean.USER_FOUND) {
     logger.debug("Account Already exists");
  }
```

This simple piece of code operates on the EventTargetBean Level attribute. This attribute is an integer flag and indicates the status of the last processing on this event. In this case, for any new account event, IGI will look to see if there's already an account-person mapping, which is the EventTargetBean.USER_FOUND value.

> This `event.getLevel()` method is used throughout the TARGET rules to check some mappings or existence of objects. Have a look at the Check level rules for the different events.

If there is already a mapping, it writes a debug message out.

Create Account [Userid Matching] Rule

This rule will attempt to match a user by the userid for the account. The code for this is:

```
when
    event : EventTargetBean(  )
    account : AccountBean(  )
then
// [ V1.5 - 2014-05-26 ]

  // Exit if account_code is null
  if (event.getCode() == null) {
    logger.info("Account code is empty. Account can not be crerated with userid
matching!");
    return;
  }
```

The first section is a data validation on the Code (userid or account id) in the event bean. If it's null, an info message is written out and the rule exits (`return;`).

```
  // Exit if already matched by a previous rule in the flow
  if (account.getPerson_id() != null) {
    logger.info("Account already matched!");
    return;
  }
```

The next section checks to see if the account (by the AccountBean) has already been matched to a person (i.e. the Person_id attribute on the account is not null). If it has been matched, the rule exits.

```
  UserBean userFilter = new UserBean();
  userFilter.setCode(event.getCode());

  BeanList beanList = UserAction.find(sql, userFilter);
  boolean found = !beanList.isEmpty();
  if (found) {
    logger.info("Account Matched by userid!");
    userFilter = (UserBean) beanList.get(0);
    account.setPerson_id(userFilter.getId());
```

The next section creates a new UserBean userFilter object, and sets the Code (which will relate to the master id or master userid on a person object) to the code (userid/accountid) from the event.

The UserAction.find will search for a user matching the filter and sets a "found" flag based on a non-empty return set. If found, an info message is written out, and the Person_id on the account is set from the first value (`beanList.get(0);`) from the found user list. There's an assumption that only one user would be found!

```
    if (account.getId() != null) {
      // the account already exist but it is unmatched/orphan
      UserAction.updateAccount(sql, account);
      logger.info("Account exist but it is unmatched/orphan");
    } else {
      UserAction.addAccount(sql, account);
      logger.info("Account : " + account.getCode() + " created!");
    }
  }
```

The last section of code is still within the found loop. It checks to see whether the account already exists or not. If it exists (and it must be an orphan as it's been through this set of rules before and wasn't' matched) it is updated (`UserAction.updateAccount();`) otherwise it is created (`UserAction.addAccount();`).

This completes this rule.

Create Account [Email Matching-] Rule

The flow of this rule is basically the same as above:

- It checks to see if the account is already matched by a previous rule (`if (accountBean.getPerson_id() != null) { return; }`).
- Pulls the email from the event bean (this is a standard attribute on the bean, with a get method; `event.getEmail();`) and if it's null it exits the flow with an info message.
- Attempts to find a matching user using the `UserAction.find()` method, where the email is a standard attribute on a userFilter (`userFilter.setEmail(email);`)
- Sets the id of the found user on the account (`account.setPerson_id();`)
- Either updates or adds a new account (`UserAction.updateAccount();` or `UserAction.addAccount();`)

The code can be seen inside IGI.

Create Account [Name-Surname Matching-] Rule

This rule is attempting to match an account to a user by serching on a firstname + surname combination. The code for this is:

```
when
    event : EventTargetBean(  )
    accountBean : AccountBean(  )
then
// [ V1.0 - 2015-06-25 ]

  // exit if already matched by a customized rule
  if (accountBean.getPerson_id() != null) {
    return;
  }
```

The first bit of code is the same as for the previous two rules – check to see if there is already a matching person for this account from a previous rule (or the previous execution of the rules for this account).

```
// name = Name
// surname = Surname

// Gets the user name,surname
String name = event.getName();
String surname = event.getSurname();

logger.info("surname = "+surname);
logger.info("name = "+name);

if (name == null || surname == null) {
  logger.info("Matching on Name, surname not applicable... exit!");
  return;
}
```

The next section of code pulls the firstname (“name” variable) and surname from the event bean. These are standard attributes on the bean and have their own get methods. The two names are written out to the info log.

The if statement checks to see if either name is null, and if so exits the rule with an info message.

```
// Look for the User into IDEAS
UserBean userFilter = new UserBean();
userFilter.setName(name);
userFilter.setSurname(surname);
BeanList ul = UserAction.find(sql, userFilter);

boolean found = !ul.isEmpty();
if (found) {
  logger.info("Account Matched by name, surname!");
  // found
  userFilter = (UserBean) ul.get(0);

  accountBean.setPerson_id(userFilter.getId()); //!!
  String eventUserCode = event.getCode();
  if (eventUserCode != null) {
    accountBean.setCode(eventUserCode);
  }

  if (accountBean.getId() != null) {
    // the account already exist but it is unmatched/orphan
    UserAction.updateAccount(sql, accountBean);
    logger.info("Account exist but it is unmatched/orphan");
  } else {
    UserAction.addAccount(sql, accountBean);
    logger.info("Account : " + accountBean.getCode() + " created!");
  }
}
```

The last section of code is similar to the earlier user searches. It builds a userFilter containing name and surname (and these are defined attributes for the user filter, so there are set methods for each) and then searches for a matching user (`UserAction.find();`).

If found, the account Person_id attribute is set to the id of the found user and the account is either updated (if it’s there but orphaned/unmatched) or added.

> This is the last of the user search rules in this flow. However you could add your own, and remove some or all of the supplied ones.

Create Account [Post Matching] Rule

The last rule is the catch-all; if the account could not be matched to a user by the rules above (i.e. by userid, by email or by firstname+surname), this rule will create an unmatched account.

```
when
              event : EventTargetBean(  )
              account : AccountBean(  )
then
  // [ V1.5 - 2014-05-26 ]
  if (account.getId() != null) {
    // Account already exists
    return;
```

The first part of the rule is the same as the others. Even if the above rules have worked (i.e. matched the account to a user) this rule will be invoked as it's in the flow. It needs a check for a prior matching and will exit if so.

```
  if (account.getPerson_id() == null) {
    // Create the UnMatched account
    String eventUserCode = event.getCode();
    if (eventUserCode != null) {
      account.setCode(eventUserCode);
    }
    UserAction.addAccount(sql, account);
    logger.info("Account created!");
    event.setTrace("Unable to match Identity!");
  }
```

This section of code will get the userid (account id) from the Code value in the event bean. It will then create an unmatched (i.e. Parent_id is null) account.

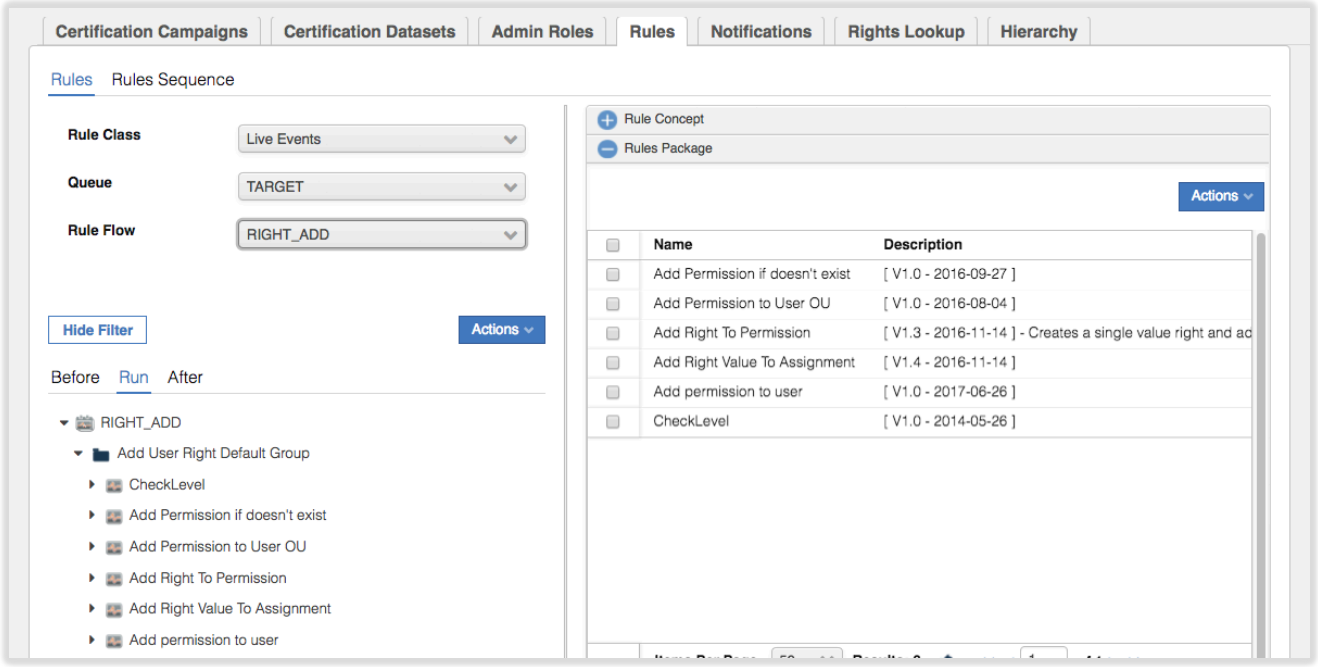
The last command sets the Trace attribute on the event to the text shown. This will appear in the Trace column of the **AGC > Monitor > TARGET Account Events** view.

This completes this rule flow.

3.3.4 Example: Process a New Rights Assignment from a Target

Rights represent a permission scope on a permission when assigned to a user (via account). If you're familiar with RACF, you can has a RACF account assigned to a RACF group with a default permission level. Similarly with Sharepoint, you might have a file access level associated with a shared file or folder.

Thus a rights object in an Add Right event represents a mapping of a right to a permission and account. The supplied rules are a good example of handling the various possible problems, such as mapping a permission or right that doesn't exist. It also shows how the users, accounts, permissions and rights are related in the data model. The flow for the Live Events/TARGET/RIGHT_ADD is shown below.



The rules are:

- **CheckLevel** – check if the account exists, the permission exists and the user is not already mapped to the permission
- **Add Permission if it doesn't exist** – if the permission in the event doesn't exist in IGI, create it
- **Add Permission to User OU** – add visibility of this permission to the OU of the account user
- **Add Right to Permission** – if the permission doesn't have this right in its list of rights values, add it
- **Add Right Value to Assignment** – add the right value to the assignment of the user to the permission
- **Add permission to user** – add the permission to the user

The following sections walk through the code.

CheckLevel Rule

The CheckLevel rule is similar to the one in the Account Create example above.

```
when
    event : EventTargetBean(  )
then
// [ V1.0 - 2014-05-26 ]
        int syncLevel = event.getLevel();
        logger.debug("Sync level: " + syncLevel);

        if (syncLevel == EventTargetBean.USER_NOT_FOUND) {
            logger.debug("Account not found");
            throw new Exception("Account " + event.getCode() + " does not exist");
        }
        if (syncLevel == EventTargetBean.PROFILE_NOT_FOUND) {
            // The following rules create the permission if it doesn't exist
            // throw new Exception("Profile not found ");
            logger.debug("Profile not found");
        }
        if (syncLevel == EventTargetBean.USER_PROFILE_FOUND) {
            logger.debug("The user already has the added profile");
        }
```

In this case it is checking that the account already exists (and it will throw an error if not found), the permission is not found (writes a debug message) and if the user is already mapped to this profile (writes a debug message).

Note that the term “profile” is used to refer to a permission.

Add Permission if doesn’t exist Rule

This rule is for data consistency – you need a permission defined before you can associate it with a user, so if the permission is not found in IGI, create it.

```
when
    event : EventTargetBean(  )
    syncBean : SyncStateBean(  )
then
if (event.getLevel() == EventTargetBean.PROFILE_NOT_FOUND) {
```

The main body of the code is a single if statement which reuses the getLevel() check from the CheckLevel rule to see if the profile (permission) is not defined to IGI.

We need this check as all rules will get executed in order irrespective of the previous rules (unless they have thrown an exception) so this rule needs to exclude the scenario where the permission already exists.

```
    // Create and publish the permission ...

    EntitlementBean entBean = syncBean.getProfile();
    // Not Administrative Role
    entBean.setAdministrative(0L);
    entBean.setExternalRef(event.getAttr3());
    if (entBean.getName() == null || entBean.getName().equals("")) {
      entBean.setName(event.getAttr3());
    }

    if (entBean.getFunctionalityType_name() == null) {
      // If Type is not setted, set equals to the application name
      entBean.setFunctionalityType_name(entBean.getApplication_name() + "Type");
    }
```

The next section of code sets up the bean for the new permission (entitlement). It will first build a new entitlement bean using the Profile on the SyncStateBean from the event.

It sets the Administrative flag to False (“0L”).

It uses Attr3 which holds the external reference (the reference on the target system, such as the group DN) as the ExternalRef attribute on the permission. If there is no name provided from the SyncStateBean for the permission, it uses the external reference.

If the Functionality Type is null, it sets it based on the Application name and the string “Type”. This is the permission type, such as “LDAPGroup” or “ADGroup”.

```
    // Create permission type if does it not exist
    FunctionalityTypeBean ft = new FunctionalityTypeBean();
    ft.setApplication_id(entBean.getApplication_id());
    ft.setName(entBean.getFunctionalityType_name());

    BeanList bl = ApplicationAction.findFunctionalityType(sql, ft);
    if (bl.size() == 0) {
      ApplicationAction.addFunctionalityType(sql, ft);
    }
```

This section creates a new funtionality type and associates it with the application (`ApplicationAction.addFunctionalityType();`).

I'm not sure why that's not in the same if block as the previous section – you would assume that if it's already there it doesn't need to be created again, but perhaps there are scenarios where it may need to be done all the time?

```
    // Create the entitlement
    try {
      entBean = EntitlementAction.insertEntitlement(sql, entBean, true);
    } catch (DBMSException e) {
      if (e.getErrorCode() == DBMSException.OBJECT_NOT_FOUND) {
        throw new Exception("Error creating permission");
      } else {
        throw e;
      }
    }
```

This try/catch block attempts to create the new entitlement in IGI and it if fails, throws an error. Its performing a database insert, so checking the DBMSException object.

```
    logger.debug("Permission " + entBean.getName() + " created, application " +
entBean.getApplication_name());

    // Update the current event object
    event.setLevel(EventTargetBean.OU_PROFILE_NOT_FOUND);
    syncBean.setProfile(entBean);

  }
```

Finally the code writes a debug message for the newly created permission.

It updates the Level of the event to say OU_PROFILE_NOT_FOUND. As the permission (profile) was not found and the rule had to create it, then there will be no permission to OU mapping, so we set that flag on the event Level attribute. We use this in the next rule. It's a good example of carrying state from one rule to another and allowing for logic flow between rules.

The last thing the code does is it sets the profile on the SyncStateBean() for the event to the newly created permission.

Add Permission to User OU Rule

This rule will publish the entitlement (permission) and add the entitlement to the OU (visibility). The code is:

```
when
    orgUnitBean : OrgUnitBean(  )
    event : EventTargetBean(  )
    syncBean : SyncStateBean(  )
then
  // [ V1.0 - 2016-08-04 ]
  // Add the profile to OU only if not already assigned
  // COMMENTED-OUT: there are some special cases in which
  // level<>OU_PROFILE_NOT_FOUND
  // even if the permission is NOT assigned to the OU
  // maybe assigned through a role)

  EntitlementBean entBean = syncBean.getProfile();
  EntitlementAction.publishEntitlement(sql, entBean);
```

In this case it is operating on the orgUnitBean, which represents the OU for the account user.

It gets the permission object from the SyncStateBean of the event and then uses this to publish the entitlement (`EntitlementAction.publishEntitlement();`). As there is no check to see whether it is already published, I assume this is harmless in that scenario.

```
  BeanList bl = new BeanList();
  bl.add(entBean);
  OrgUnitAction.addRoles(sql, orgUnitBean, bl, false);

  String pName = entBean.getName();
  String oName = orgUnitBean.getName();

  logger.debug("Profile " + pName  + " assigned to OU " + oName);

  // Update the current event object
  event.setLevel(EventTargetBean.USER_PROFILE_NOT_FOUND);
```

Next it adds the entitlement to the OU (`OrgUnitAction.addRoles();`).

It gets the permission name (pName) and OU name (oName) from the different beans and writes out a debug message with them. Finally it sets the event level to USER_PROFILE_NOT_FOUND. This is setting the level to say that there is no existing mapping between the account user and this permission.

Add Right To Permission Rule

This rule will add the right (as a single value right) to the permission if it is not already defined. Recall that in the data model, a set of rights values is associated with a permission (and then a specific rights value is associated with the user-permission assignment).

```
when
    userBean : UserBean(  )
    event : EventTargetBean(  )
    syncBean : SyncStateBean(  )
then
// [ V1.3 - 2016-11-14 ]

  EntitlementBean profile = syncBean.getProfile();
```

The rule is using the UserBean, EventTargetBean and SyncStateBean.

As with previous rules in the flow, this one retrieves the object for the permission from the event data (syncBean.getProfile();).

```
RightsDAO rightsDao = new RightsDAO(logger);
rightsDao.setDAO(sql);

ServiceAttributeBean sAB = new ServiceAttributeBean();
sAB.setName(event.getAttr5());
```

It creates a new RightsDAO (Data Access Object) and sets it to “sql”.

Next it creates a new ServiceAttributeBean and sets the name of it to ATTR5. ATTR5 on a RIGHT_ADD is the right name for the permission. For example it could be that the permission (profile) is fileshare “Accounts_ABC” and the right value is “AccessLevel”. The ServiceAttributeBean name in this case would be “AccessLevel”.

```
if (profile.getAttributeBased() != null && profile.getAttributeBased() > 0) {
  // attribute based permissions don't have lookup
} else {
  String key = profile.getFunctionalityType_name() + "_" + event.getTarget() + "_" +
event.getAttr5();
  sAB.setContent(key);
}
```

In IGI we have two types of permissions; the traditional group-based (like AD Groups) and attribute-based permissions (like RACF User flags for SPECIAL and OPERATIONS). The traditional group-based permissions provide lookups via rights lookup lists (i.e. lists of values associated with an IGI “Right” that is attached to a permission). The attribute-based permissions have a set of rights values directly associated with the attribute-permission definition.

This bit of code is checking to see if its an attribute-based permission, and if it isn’t, then it builds the ServiceAttributeBean content to a key of <permission tye>_<application>_<right name>. For example if the permission type was “SharedFolder” and the application was “FileServerHomer”, then the key will be “SharedFolder_FileServerHomer_AccessLevel”.

```
BeanList result = rightsDao.findProfileRights(sAB, profile, null);
if (!result.isEmpty()) {
  return;
}

sAB.setMultiple(1L);
sAB.setLookup(1L);

BeanList rightsList = new BeanList();
rightsList.add(sAB);

rightsDao.insertOrUpdateRight(rightsList, profile);
```

The next bit of code will search the current list of rights for the permission (`rightsDao.findProfileRights();`) for a rights value matching. If it finds one it exits the rule (`return;`).

Otherwise it continues and sets the Multiple and Lookup flags to true (`1L`), builds a new array (BeanList) and adds the new rights value to it, then performs an add or mod (`rightsDao.insertOrUpdateRight(rightsList, profile);`) for the rights value on the existing list (if there) on the permission.

This concludes the rule.

Add Right Value to Assignment Rule

This rule will add the new right value to the user-permission assignment.

```
when
    userBean : UserBean(  )
    event : EventTargetBean(  )
    syncBean : SyncStateBean(  )
then
// [ V1.4 - 2016-11-14 ]
    // If user does not have the permission, assign to user

    EntitlementBean profile = syncBean.getProfile();
```

The first part of the rule is the same as the previous rule; it uses the same beans and gets the permission (profile) from the event syncBean.

```
    if (profile.getAttributeBased() != null && profile.getAttributeBased() > 0) {
            // attribute based permissions don't have lookup
       } else {
            String key = profile.getFunctionalityType_name() + "_" + event.getTarget() +
"_" + event.getAttr5();
            RightsAction.insertServiceAttrLookup(sql,key,event.getAttr6());
       }


UserAction.addServiceAndPermission(sql,userBean,profile,event.getAttr5(),event.getAttr6())
;
```

The next bit of code has the same check for an attribute-based permission. If it is not an attribute-based permission, it builds the ServiceAttributeBean content to a key of <permission tye>_<application>_<right name>. For example if the permission type was “SharedFolder” and the application was “FileServerHomer”, then the key will be “SharedFolder_FileServerHomer_AccessLevel”.

It then creates a rights value lookup list (i.e. the IGI Rights definition) with a single value (`RightsAction.insertServiceAttrLookup();`). The right value is in Attr6. For example, Attr5 may be “AccessLevel” and Attr6 may be “READ”.

Finally, it adds the right and permission to the user with the `UserAction.addServiceAndPermission();` method.

Add permission to user Rule

This rule is a bitch of a catch-all to make sure any conditions raised in earlier rules have been processed. It is a series of "if this condition then do this" blocks to capture any outstanding conditions.

```
when
    userBean : UserBean(  )
    orgUnitBean : OrgUnitBean(  )
    event : EventTargetBean(  )
    syncBean : SyncStateBean(  )
then

  if (event.getLevel() == EventTargetBean.USER_NOT_FOUND) {
    throw new Exception("Account " + event.getCode() + " does not exist");
  }
```

The first section of code is checking to see if the account user is not found (i.e. doesn't exist). This is redundant as it's checked in the CheckLevel rule and if `EventTargetBean.USER_NOT_FOUND` is matched, the code threw an error.

```
  EntitlementBean entBean = syncBean.getProfile();

  String pName = entBean.getName();
  String uCode = userBean.getCode();

  // User already has the profile
  if (event.getLevel() == EventTargetBean.USER_PROFILE_FOUND) {

    logger.debug("Permission " + pName + " already assigned to User " + uCode);

    return;
  }
```

The next block of code is checking to see if the user-permission assignment for the permission in the new right is there. If the previous rules have worked successfully, the permission (and associated right) should have been assigned to the user. If it's there, the rule exits.

```
  // User does not have the profile
  if (event.getLevel() == EventTargetBean.USER_PROFILE_NOT_FOUND) {

    BeanList bl = new BeanList();
    bl.add(entBean);
    UserAction.addRoles(sql, userBean, null, bl, true, false);

    logger.debug("Permission " + pName + " assigned to User " + uCode);

    return;

  }

  throw new Exception("Unexpected sync level: " + event.getLevel());
```

This last block is checking to see if the user to permission assignment for the permission in the new right is not there. This should not occur if the previous rules were successful, but may occur if the right name was null. In this case it will add the permission to the user (`UserAction.addRoles();`) and exit the rule.

If the bottom of the rule is reached, then there is some other untrapped error, so throw an exception.

Summary of Rule Flow for the RIGHT_ADD Event

This is a complex rule flow, comprising six discrete rules. The flow is:

- Check the condition code (event Level) on entering the rule flow. If the account user is not found, an exception is thrown and the rule flow ends. For some other conditions, some debug messages are written, but the flow continues for the event.
- The business logic:
    - Determine the permission (profile) for this right, and if it doesn't exist, add it and flag a condition code saying the OU-Permission mapping (visibility) is not set.
    - Add the new permission to the users OU, and flag a condition code saying the user-profile mapping is not set
    - Add the new right value directly to the permission (and if it's not an attribute-permission, create a new IGI Right name for the right value for use in the right-permission mapping)
    - Add the new right value to the user-permission mapping (and if it's not an attribute-permission, create a new IGI Right for the right value)
- Check the condition codes (event Level) for any outstanding conditions. It checks for account user not found (but this should throw an error in the first rule), user has permission (just exits) and user does not have permission (adds permission, without rights, to the user).

This set of rules is a good example of handing complex inter-related objects (user, permission, rights) and their assignments, and allowing for different error conditions that may arise with the different objects and relations.

3.3.5 Example: New Permission Assignments Drive a Continuous Campaign

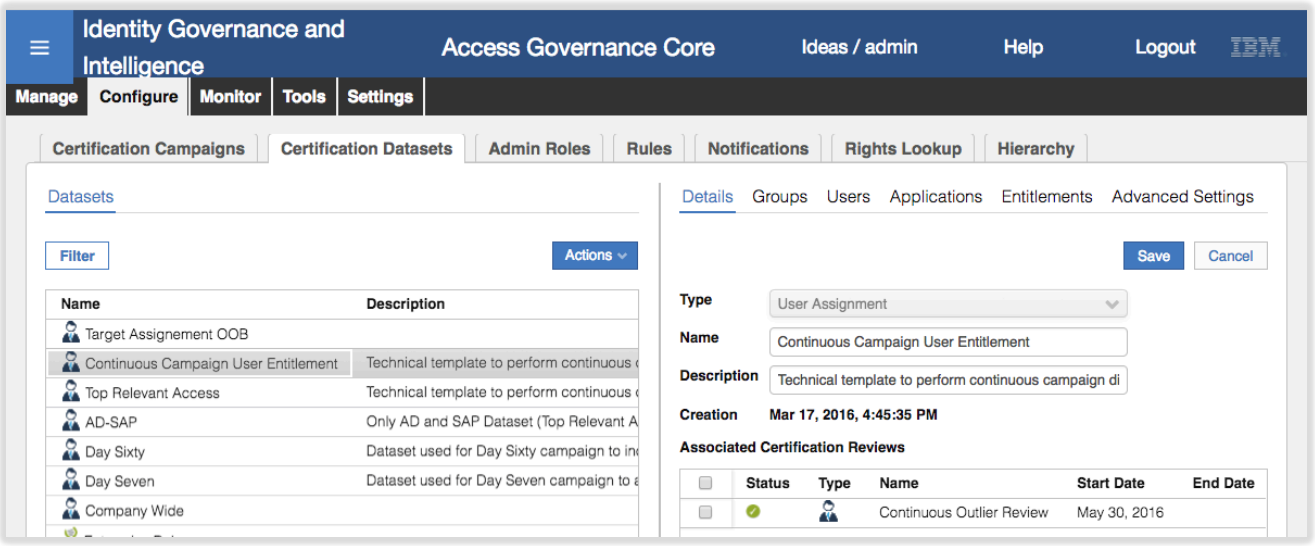
The last example for the Live Events/TARGET rules is an example of using a new permission assignment (Live Events/TARGET/PERMISSION_ADD) to trigger a access review in a continuous certification campaign. This is similar to the example in Example: User Move Triggers Continuous Certification Campaign on page 38.

```
when
    userBean : UserBean(  )
    event : EventTargetBean(  )
    syncBean : SyncStateBean(  )
then
```

The rule, like most of the TARGET queue rules, uses the EventTargetBean (describing the event itself) and the SyncStateBean (containing specifics about the permission). It also uses the UserBean for the account user.

```
  String targetApplicationContentToRecertify = "PadLock";
  String templateNametoUse = "Continuous Campaign User Entitlement";

  // Check if we have the right Application
  if (!event.getTarget().equalsIgnoreCase(targetApplicationContentToRecertify) ) {
    logger.info("Target is not " + targetApplicationContentToRecertify + ", skip");
       return;
  }
```

The next bit of code sets two hardcoded string variables; the application name (in this case "PadLock") and the continuous campaign dataset name (in this case "Continuous Campaign User Entitlement") as shown below.



There is a check to see if the target (application) in the event bean matches the one we're looking for ("PadLock") and if not, if logs an info message and exits the rule (`return;`).

```
// Create the Template Bean
TemplateBean templateBean = new TemplateBean();
templateBean.setName(templateNametoUse);

// Create the DAO to find the DataSet
TemplateDAO templateDAO = new TemplateDAO(logger);
templateDAO.setDAO(sql);

// Find the requested template to use
BeanList blTemplateBean = templateDAO.find(templateBean, new Paging(4));

if(blTemplateBean.size()==0){
  throw new Exception("Template does not exists");
}
```

The next bit of code creates a new TemplateBean (certification dataset) and sets the name to the hardcoded name at the top of the rule ("Continuous Campaign User Entitlement").

It creates a new template data access object (`TemplateDAO();`) and sets it to an SQL type data access object.

This is used to search for a campaign dataset matching that name (`templateDAO.find();`). If not found (`blTemplateBean.size()==0`) then it throws an exception which will exit the rule and flow.

```
// Link to the found Template
templateBean = (TemplateBean) blTemplateBean.get(0);

// Create the Bean List to add to the Continuos Campaign
AbstractBean[] element = new AbstractBean[2];
element[0] = userBean;
element[1] = syncBean.getProfile();

BeanList listBean = new BeanList();
listBean.add(element);
```

```
  // Add To the Campaign
  if (listBean.size() > 0) {
    templateDAO.addEntity(listBean, AttestationRes.TEMPLATE_ENTITY_USERENT, templateBean,
AttestationTypes.PERSON_ENTITLEMENT.getValue());
  }
```

The last block of cide will link the template object with the found template.

It creates an array with two elements (`new AbstractBean[2];`) and sets the pair of values to the user bean (i.e. the user for this account) and the permission (profile from the event data). A new `BeanList` (array) is defined and the single pair (user object and permission) are added to it (`listBean.add(element);`).

Finally this new BeanList (i.e. user-permission pair) is added to the campaign dataset (`templateDAO.addEntity();`) and will be available to the next campaign reviewer to log into the service center. The method includes specifying the campaign dataset type (`AttestationRes.TEMPLATE_ENTITLE_USERENT`) and attestation (campaign) type (`AttestationTypes.PERSON_ENTITLEMENT`).

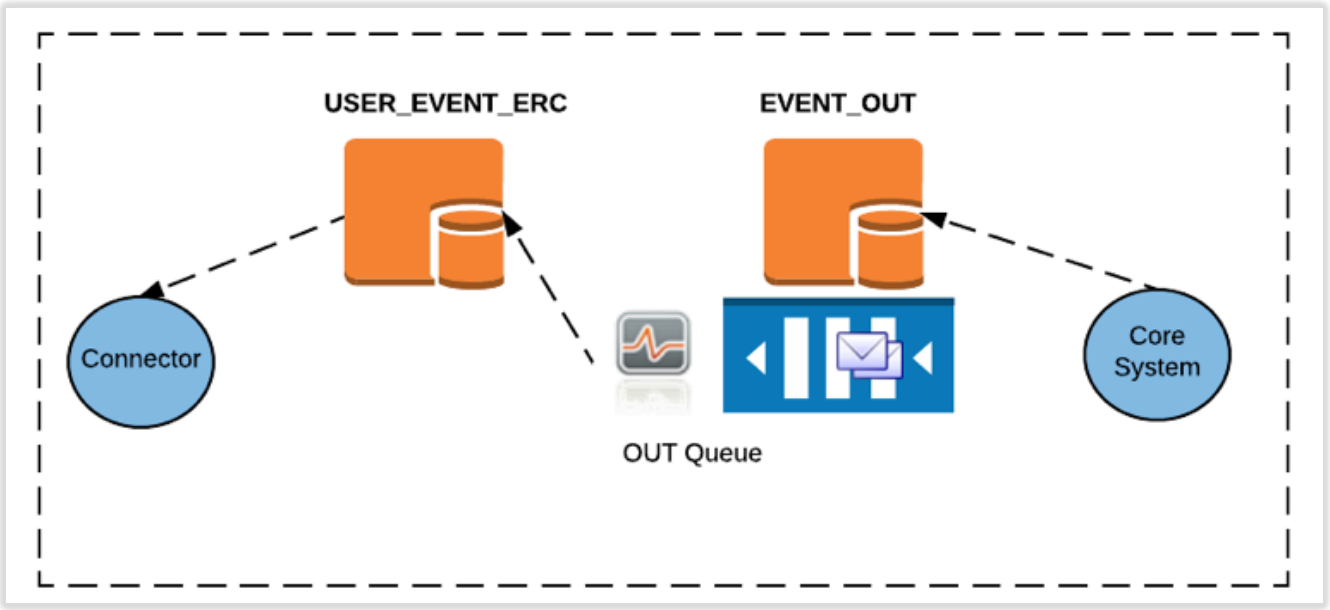
This concludes the code in this example.

3.4 Processing OUT Account and Permission Events

The OUT queue is used to process outgoing account and permission events (i.e. the provisioning activity). This section looks at the flows, events and rules that can be applied.

3.4.1 Event Flows and Events for the OUT Queue

The OUT event flow is shown in the following figure.



Activity in IGI, such as access requests by users or system generated account and access changes, will cause Account and Permission events to be written to the EVENT_OUT table (the OUT queue). These events are processed by rules and written to the USER_EVENT_ERC table. This table is accessed by the Enterprise Connector framework (either a legacy Enterprise Connector or a Brokerage Adapter) or other event consumers, for provisioning account and access changes to target systems. When the provisioning activity is complete, the consumer will update the event in the USER_EVENT_ERC table with status and messge information.

Rules can operate on the events in the OUT queue. They are only processed as Live Events (there are no Deferred Events for the TARGET queue).

Input Events	Rule Class	Queue	Rule Flow	Notes
Create Account	Live Events	OUT	ACCOUNT_CREATE	
Modify Account	Live Events	OUT	ACCOUNT_MODIFY	
Change Password	Live Events	OUT	ACCOUNT_PWDCHANGE	
Disable User	Live Events	OUT	ACCOUNT_DISABLE	1., 2.
Enable User	Live Events	OUT	ACCOUNT_ENABLE	1., 2.
Remove Account	Live Events	OUT	ACCOUNT_REMOVE	
Add Permission	Live Events	OUT	PERMISSION_ADD	
Remove Permission	Live Events	OUT	PERMISSION_REMOVE	
Add Right	Live Events	OUT	RIGHT_ADD	
Remove Right	Live Events	OUT	RIGHT_REMOVE	
Add Delegation	Live Events	OUT	DELEGATION_ADD	
Remove Delegation	Live Events	OUT	DELEGATION_REMOVE	

Add Resources	Live Events	OUT	RESOURCE_ADD	
Remove Resources	Live Events	OUT	RESOURCE_REMOVE	
Add Entitlement to User	Live Events	OUT	USERROLE_ADD	1.
Remove Entitlement from User	Live Events	OUT	USERROLE_REMOVE	1.
	Live Events	OUT	BEFORE	3.

NOTES:

1. There is some confusion with the use of "user" with respect to the OUT queue. The OUT queue will only ever process accounts, but some systems (like SAP) call their accounts "users". Do not confuse this with users (People) in the IN queue events. Luckily the rule flows are called Account.
2. The Enable/Disable mechanism is supported in IGI, however not all target systems have a active/inactive flag and not all adapters support setting such a flag.
3. There is a common BEFORE rule that can run before any other, event-specific, rule. This is like a common entry point and can be used to conserve rules coding/management (rather than replicating some code across multiple rules, put it in the BEFORE event flow).

Thus, the OUT queue is processing events for various objects being provisioned from IGI:

- **Accounts** – processing incoming add, modify and delete events. This includes password management and enable/disable events sent from IGI.

- **Permissions** – a permission, in the context of a OUT queue event, is the assignment of an account to a target system permission. IGI does not expose any means to create a permission on a target system, only manage membership (if you are familiar with ISIM, it can create/delete permissions via some adapters like AD).

- **Rights** – a mapping of a right on a permission for a user. For example a RACF account may be mapped to a RACF group with a default access permission (a "right" in IGI terminology) of READ.

- **Delegations** – TBC

- **Resources** – a resource can be some other (non-access) object associated with a user and provisioned through the same mechanism. These are rarely used.

- **User Roles** – most systems will no process User Roles. The exception is ISIM, via the ISIGADI integration. There are some rules that operate on userrole events.

Most events passing through the OUT queue will relate to accounts and permissions. For example, if a user requests some AD groups but they don't already have an AD account, IGI will generate two types events:

- A "Create Account" event – for the new AD account, and
- A series of "Add Permission" events – one for each AD account to AD group mapping requested

3.4.2 Objects Available for OUT Events

The following beans may be available in the working memory (based on the event):

- *EventBean()* / *EventOutBean()* – common event details (like operation type)
- *UserBean* – user object (i.e. user for this account/permission mapping/right mapping)
- *ExternalInfo* – additional user attributes
- *AccountBean* – account object
- *OrgUnitBean* – OU object

These can be found in the JavaDoc but aren't very well defined. Not every bean will be defined for every event type, so you need to be careful if expecting beans to be available (e.g. processing an account event and expecting there to be a person associated).

The *EventBean()* (which has a subclass of EventOutBean()) includes an operation attribute (getOperation()) that is a numeric representation of the operation the event represents, including:

- 1: PERMISSION_ADD
- 2: PERMISSION_REMOVE
- 4: DELEGATION_ADD
- 5: DELEGATION_REMOVE
- 6: ACCOUNT_DISABLE
- 7: ACCOUNT_ENABLE
- 8: ACCOUNT_CREATE
- 9: ACCOUNT_REMOVE
- 10: ACCOUNT_MODIFY
- 11: ACCOUNT_PWDCHANGE
- 12: RIGHT_ADD
- 13: RIGHT_REMOVE
- 14: RESOURCE_ADD
- 15: RESOURCE_REMOVE
- 16: USERROLE_ADD
- 17: USERROLE_REMOVE

The complete list an be found at
https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/db_tables/ref/db_EVENT_OUT.html

The rules can also access other objects in IGI, such as roles/permissions and accounts, as shown in the following examples.

3.4.3 OUT Event Rules vs. EC Mapping Rules vs. Adapter Logic

For account and permission events flowing out from IGI to a target system, there are multiple places where custom business logic can be injected;

- **The OUT Event Rules** – this is a common set of rule flows for each event type. So all events irrespective of target or target platform type will go through the rule flow

- **The Enterprise Connector framework Mapping Rules** – each connector definition (whether it represents a legacy Connector or Brokerage Adapter) will have pre- and post-mapping rules. These will be unique for each target. For example if you have two AD systems you are provisioning through, you will have two Enterprise Connector definitions, each which can have their own mapping rules.

- **The Adapter Logic** – most adapters are written on Directory Integrator, thus can be modified (out-of-the-box adapters should not be modified as later versions may overwrite) or cloned as custom adapters. Thus custom business logic can be built into the adapter code. Any changes will apply to every instance of the adapter of the same type (e.g. if you modify the LDAP Adapter, every adapter of type LDAP will use those changes).

Where should you apply custom business logic? It depends on the logic you want to apply. If you want it to apply to every event of a type (e.g. Create Account), then using the OUT Event Rules makes sense – rather than coding logic into every EC Mapping Rule or customising adapters.

If you need to use the events to make changes to other objects in IGI, then you are better off using OUT Event Rules. You may have limited access into IGI objects from the EC Mapping Rules or custom adapters, or it may be more complex to code (e.g. using EJB or REST APIs).

If you have business logic that only applies to a specific target, then using the EC Mapping Rules makes more sense. You could do it in the OUT Events Rules, but would need a lot of switching logic for different targets making maintenance more complex.

Creating adapters is a more complex piece of work, so it would be preferable to code business logic into the OUT Event Rules or EC Mapping Rules. They also provide more visibility into custom business logic; you can view the code in the IGI Admin Console, whereas if it's coded in a TDI assembly line, you need to extract the definition out of IGI and load it up into the Configuration Editor to see the logic.

3.4.4 Example: New Permission Mapping Drives Continuous Campaign

As in earlier examples, we can dynamically put user entitlements into a continuous certification campaign. Previous examples can be seen above in Example: User Move Triggers Continuous Certification Campaign (page 38), and Example: New Permission Assignments Drive a Continuous Campaign (page 57).

In this case we have an Add Permission rule (Live Events/OUT/PERMISSION_ADD/Intercept Permission and Certify User). It may be that you want any added permission to feed into a continuous certification campaign, irrespective of whether its come from an access reuqest in the Service Center, driven by an admin in the Admin Console, or internally generated. The OUT queue is the only common point for these different actions.

The code is very similar to the code in the earlier examples.

```
when
    userBean : UserBean(  )
    orgUnitBean : OrgUnitBean(  )
    eventOut : EventOutBean(  )
then
// [ V1.0 - 2015-11-05 ] by LL
  BeanList entitlements = UserAction.findJobRoles(sql, userBean);

  TemplateDAO templateDAO = new TemplateDAO(logger);
  templateDAO.setDAO(sql);

  TemplateBean templateBean = new TemplateBean();
  templateBean.setName("CertifyNewHiresDataset");

  String permissionToCheck = eventOut.getValore1();
```

```
  BeanList blTemplateBean = templateDAO.find(templateBean, new Paging(4));
```

It builds a list of entitlements for the user (`UserAction.findJobRoles();`)

It sets up a new template (cert dataset) data access object and uses the title of "CertifyNewHiresDataset". It gets the permission from the event (note the old method `eventOut.getValore1();`).

It will try to find the cert dataset (`templateDAO.find();`) as in the earlier examples.

```
  if (permissionToCheck !=null &&
permissionToCheck.equalsIgnoreCase("dummyDemoPermission")) {
```

The next section of code will check that the permission is not null and matches a specific permission name (this was done for a customer demo). You wouldn't normally restrict down to a specific permission, but you may restrict to a specific application.

```
  if(blTemplateBean.size()==0){
    throw new Exception("Template does not exists");
  }
  templateBean = (TemplateBean) blTemplateBean.get(0);

  BeanList listBean = new BeanList();
  for (int i = 0; i < entitlements.size(); i++) {
    AbstractBean[] element = new AbstractBean[2];
    element[0] = userBean;
    element[1] = (EntitlementBean) entitlements.get(i);
    listBean.add(element);
  }
  if (listBean.size() > 0) {
    templateDAO.addEntity(listBean, AttestationRes.TEMPLATE_ENTITY_USERENT, templateBean,
AttestationTypes.PERSON_ENTITLEMENT.getValue());
  }

  }
```

If so, it will check that the certification dataset was found. If not it throws an exception and exits.

Otherwise it builds a beanlist of user and permission mappings (unlike earlier examples, this code allows for multiples) which is added to the certification dataset, as we saw in the earlier examples.

3.5 Processing OUT User Events

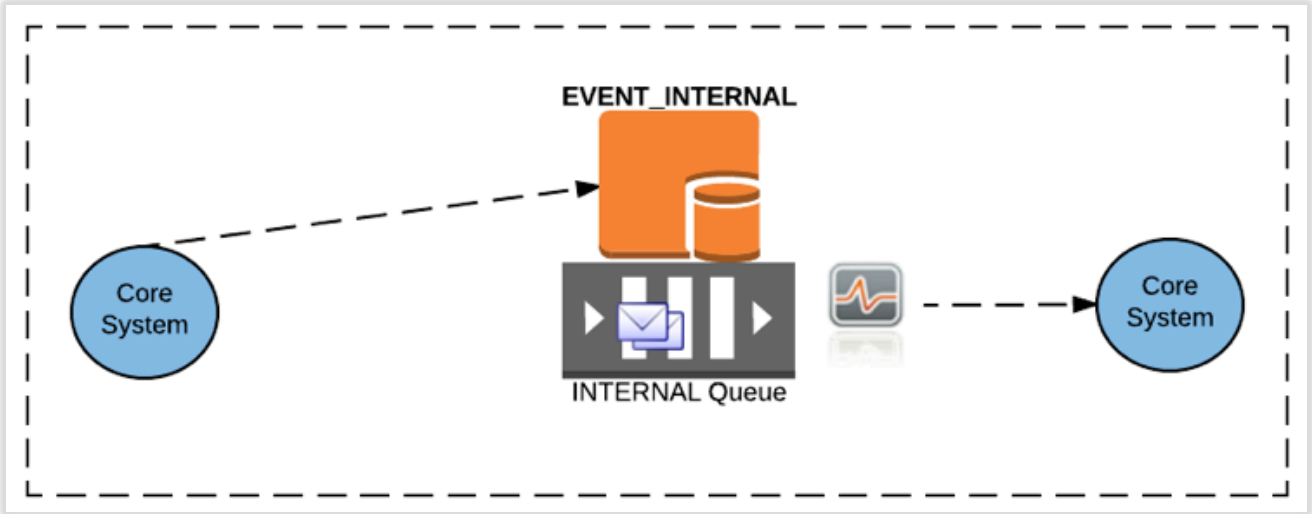
IGI 5.2.3.1 (Fixpack 1 for IGI 5.2.3) introduced another OUT queue to process user changes. This was done specifically for the ISIM-IGI integration module (ISIGADI) and should not be used for rules processing at this time.

3.6 Processing INTERNAL Events

The INTERNAL queue is used to process internal events, such as application and org unit changes within IGI. This section looks at the flows, events and rules that can be applied.

3.6.1 Event Flows and Events for the INTERNAL Queue

The INTERNAL event flow is shown in the following figure.



Various operations wihin IGI can write events to the EVENT_INTERNAL table (INTERNAL queue) and rules can process these events as we've seen with other queues.

The Internal Queue is not enabled by default, it must be enabled for events to appear in the queue.

Rules can operate on the events in the INTERNAL queue. They are only processed as Live Events (there are no Deferred Events for the TARGET queue).

Input Events	Rule Class	Queue	Rule Flow
Create Application	Live Events	INTERNAL	APPLICATION_CREATE
Modify Application	Live Events	INTERNAL	APPLICATION_MODIFY
Remove Application	Live Events	INTERNAL	APPLICATION_DELETE
Create Entitlement	Live Events	INTERNAL	ENTITLEMENT_CREATE
Modify Entitlement	Live Events	INTERNAL	ENTITLEMENT_MODIFY
Remove Entitlement	Live Events	INTERNAL	ENTITLEMENT_DELETE
Create User	Live Events	INTERNAL	USER_CREATE
Modify User	Live Events	INTERNAL	USER_MODIFY
Remove User	Live Events	INTERNAL	USER_DELETE
Create OU	Live Events	INTERNAL	ORGUNIT_CREATE
Modify OU	Live Events	INTERNAL	ORGUNIT_MODIFY
Remove OU	Live Events	INTERNAL	ORGUNIT_DELETE
Change SOD Status	Live Events	INTERNAL	<not yet implemented>
Add User Entitlement	Live Events	INTERNAL	<not yet implemented>
Remove User Entitlement	Live Events	INTERNAL	<not yet implemented>
	Live Events	INTERNAL	BEFORE

Thus, the INTERNAL queue is processing events for various objects following changes within IGI:

- **Applications** – application definitions
- **Entitlements** – both internal (IT and Business Roles) and external (External Role and Permissions)
- **Users** – people in IGI
- **OrgUnits** – OU changes

Use of rules on the INTERNAL queue objects can be very useful for triggering other internal actions.

3.6.2 Objects Available for OUT Events

The following beans may be available in the working memory (based on the event):

- *EventBean() / EventInternalBean()* – common event details (like operation type)
- *ApplicationBean* – application object
- *EntitlementBean* – entitlement object
- *UserBean* – user object (i.e. user for this account/permission mapping/right mapping)
- *ExternalInfo* – additional account objects
- *OrgUnitBean* – OU object

These can be found in the JavaDoc but aren't very well defined. Not every bean will be defined for every event type, so you need to be careful if expecting beans to be available.

The operations for the INTERNAL queue are not documented. However there is an EntityType attribute on the EventInternalEntityType object to tell the object type:

- `EventInternalEntityType USER`,
- `EventInternalEntityType ORGUNIT`,
- `EventInternalEntityType ENTITLEMENT`,
- `EventInternalEntityType APPLICATION`,
- `EventInternalEntityType TASK`, `EventInternalEntityType ACCOUNT` and
- `EventInternalEntityType GROUP`

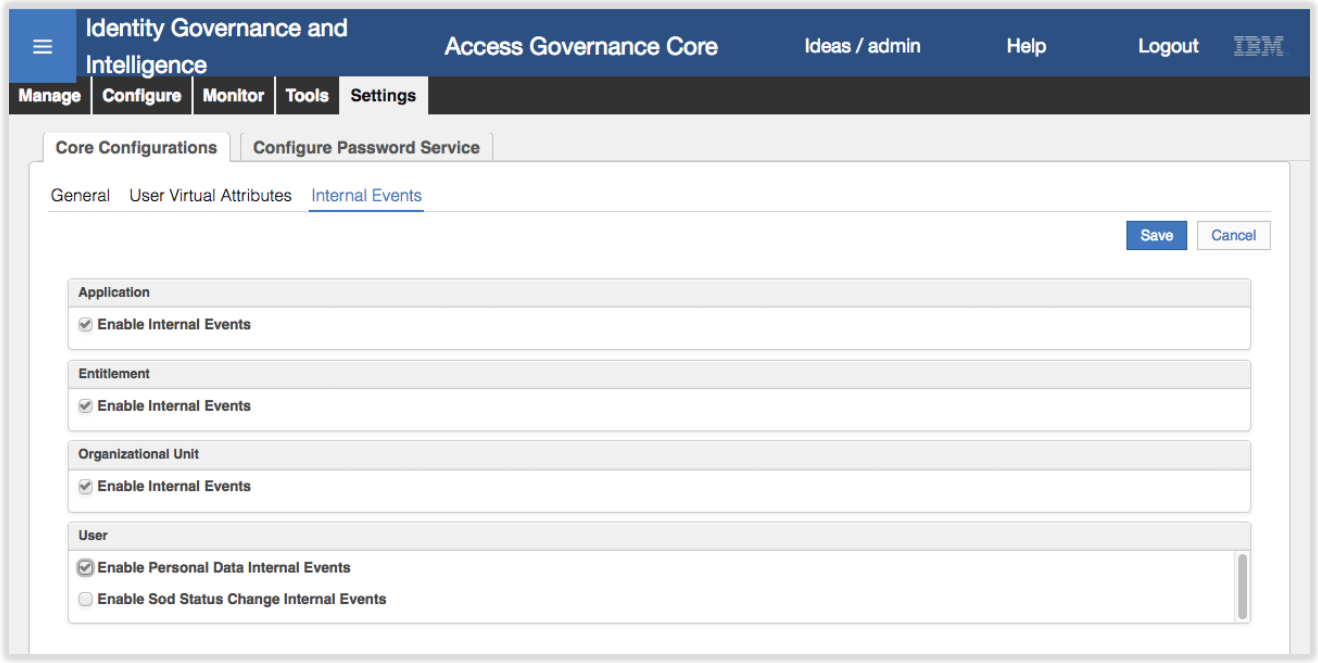
(this is from the JavaDoc in the SDK and shows more event types than are exposed in the Rules UI).

The next section describes how to enable the INTERNAL queue and then some examples follow.

3.6.3 Enabling the INTERNAL Queue

There are two things to be done: Enable the Internal Events and set a BEFORE rule to make sure the events stay on the queue after they are processed.

The INTERNAL queue is disabled by default. It can be enabled in **Access Governance Core > Settings > Core Configurations > Internal Events**.

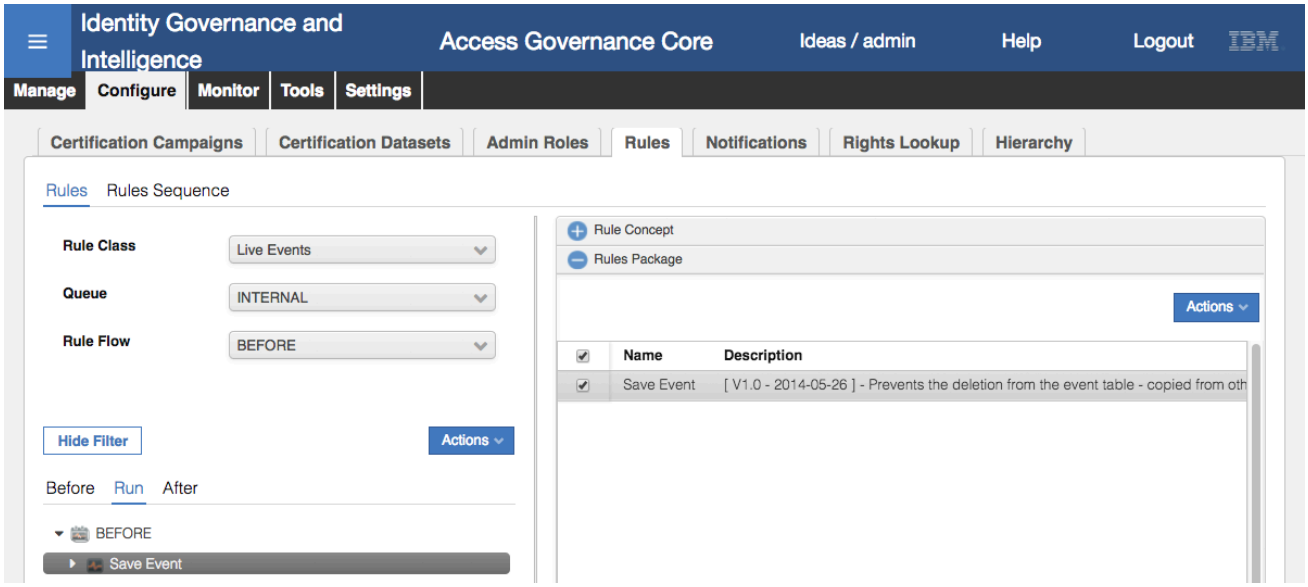


The different internal events (Application, Entitlement, OU and User) can be individually enabled. This will allow the INTERNAL queue rules to run.

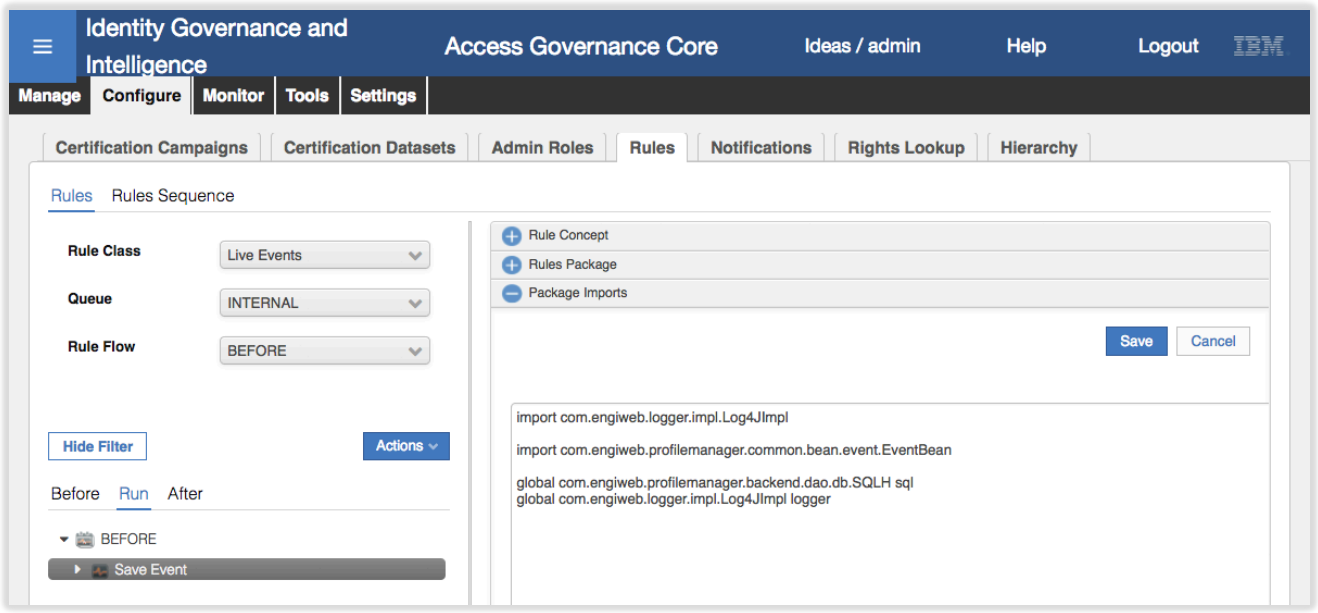
However the product, by default, won't show the events in the **AGC > Monitor > INTERNAL events** view. To resolve this, you need to build a Live Events/INTERNAL/BEFORE rule that will force the events to hang around. In the Rules Package pane, select **Actions > Create** and enter rule details like below (this has been copied from similar rules for BEFORE flows on other queues).

The code uses the EventBean and sets the state to True (`1L`). This means it will stay on the INTERNAL queue after IGI has processed it.

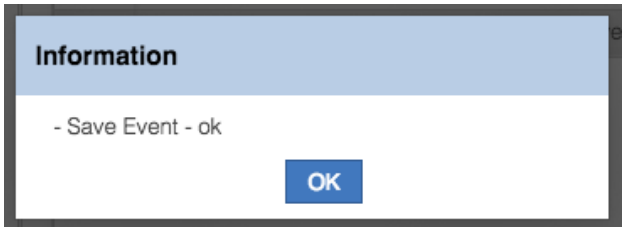
Save this rule, and add it to the Live Events/TARGET/BEFORE event (select both the flow and the new rule in the Rule Package and **Actions > Add**).



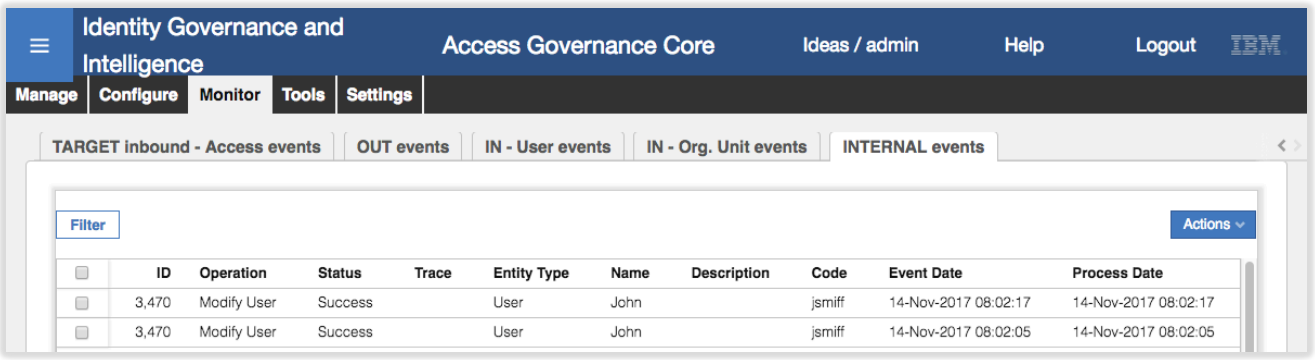
You also need to make sure you have the bean accessible by adding the relevant class (`import com.engiweb.profilemanager.common.bean.event.EventBean`) to the Package Imports from the other BEFORE rule.



As with any new rule, you should verify it once you have the Package Imports set. Go back to the Rules Package, select the new rule and use **Actions > Verify**. You should get a success dialog.



With the INTERNAL events enabled, and the new Save Event rule in place, the next cycle of the RuleEngine task should show any object change events in **AGC > Monitor > INTERNAL Events**.



This confirms the INTERNAL queue is enabled and working.

3.6.4 Example: Internal Queue Rule - TBA

To be added.

4 Data Mapping Rules for Enterprise Connector Flows

The previous sections have looked at rules for the various event queues (IN, OUT, TARGET and INTERNAL). Associated with many of these flows are the Enterprise Connectors (and Brokerage Adapters) that can feed events into the IN/TARGET Queues and consume events from the OUT Queue(s). This section looks at the Enterprise Connector flows and how rules can be applied to perform custom data mapping.

4.1 Enterprise Connector Framework, Connectors / Adapters and the Queues

We have discussed in previous sections have we can use the queues and associated rules to enhance and extend the event data flows:

- The IN queue and rules process changes to users and OUs being fed into IGI (such as from a HR Feed),
- The TARGET queue and rules process changes to accounts and permissions being fed into IGI (such as from a reconciliation),
- The OUT queue and rules process changes to accounts and permissions being pushed out from IGI to a target (provisioning), and
- The INTERNAL queue and rules can process internal events such as OU and application changes

The IN / OUT / TARGET queues provide the external interface to IGI for normal processing (ignoring bulk loads and API operations). Any external system could write events to, or read events from, the IGI queues.

The Enterprise Connector framework is the module for managing both the legacy Enterprise Connectors (from CrossIdeas) and the Identity Brokerage Adapters (ISIM heritage). The Enterprise Connector framework is an IGI module that runs as part of the IGI application (in the same Liberty profile and using the same JVM) but acts as an external component and reads from or writes to the queues. For example, a HR adapter (like the SAP HR Adapter) will push user and OU events into IGI via the Enterprise Connector framework and the IN queue. Similarly adapters and connectors can pull/push account and permission events via the queues and the Enterprise Connector framework.

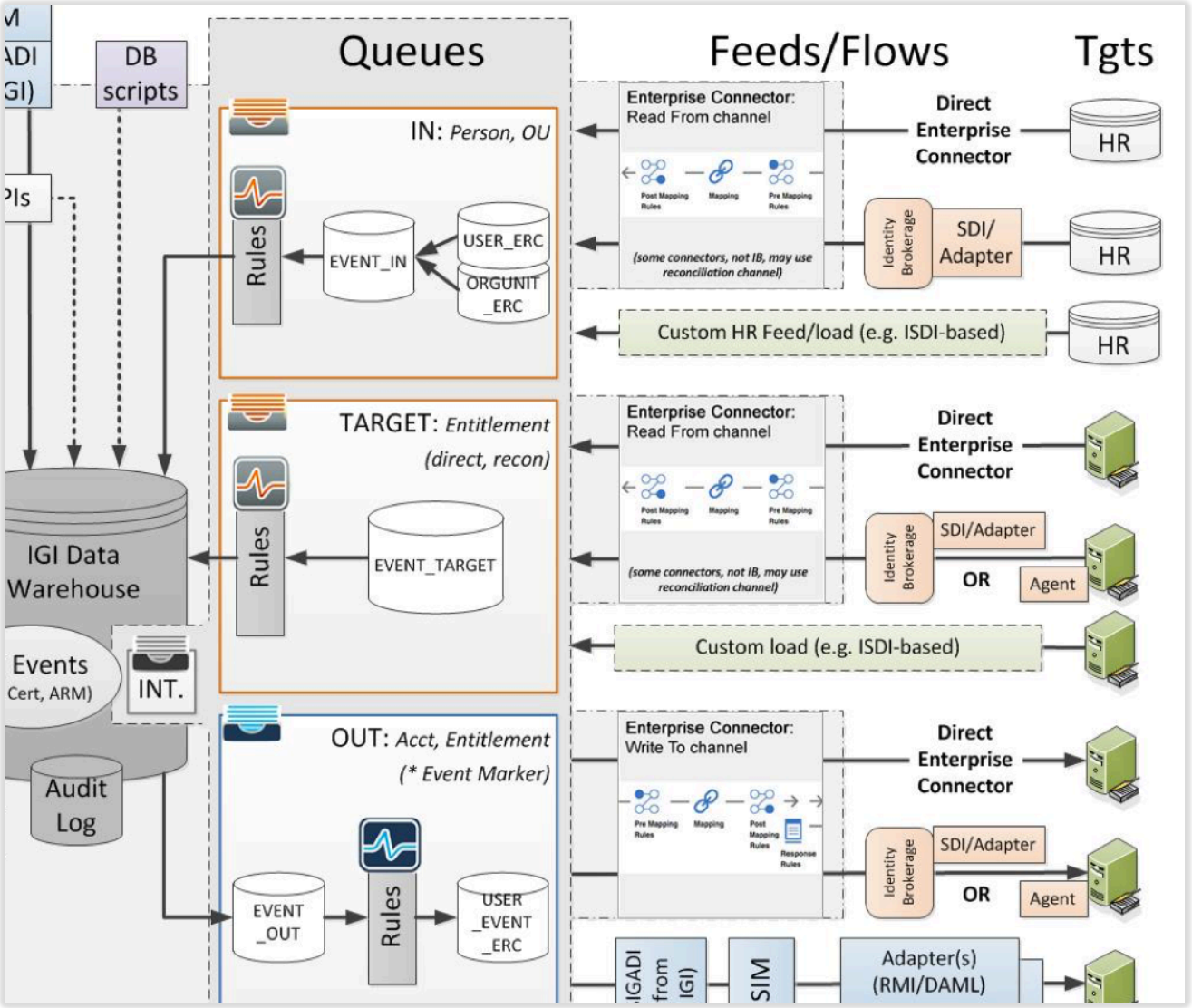
The following figure is a partial view of an IGI data flow diagram showing the queues, rules and Enterprise Connectors. It shows:

- The user/OU events flowing from HR systems, via either a legacy Enterprise Connector or an Identity Broker, through the Enterprise Connector framework and onto the USER_ERC or ORGUNIT_ERC table to be process by the Live (or Deferred) Events/IN/<event> rule flows

- The account/permission events flowing from target systems, via either a legacy Enterprise Connector or an Identity Broker, through the Enterprise Connector framework and onto the EVENT_TARGET table to be process by the Live Events/TARGET/<event> rule flows, and

- The account/permission events flowing out of IGI, through the EVENT_OUT table, processed by the Live Events/OUT/<event> rule flows, to the USER_EVENT_ERC table, which the Enterprise Connector framwork will process for the legacy Enterprise Connectors or Identity Brokerage Adapters to provision to the target systems

It also shows the INTERNAL queue. Ignore the other components shown as they are not relevant to the rules discussion.



As we have seen in the last few sections, we can apply business logic to events as they arrive on the queues. We can also apply business logic to the data flowing through the Enterprise Connector framework and we will cover this in the next few sections.

4.2 Connector Channel Modes and Data Mapping

The Enterprise Connector framework has the concept of Channel Modes. A connector (or adapter) can run in multiple modes.

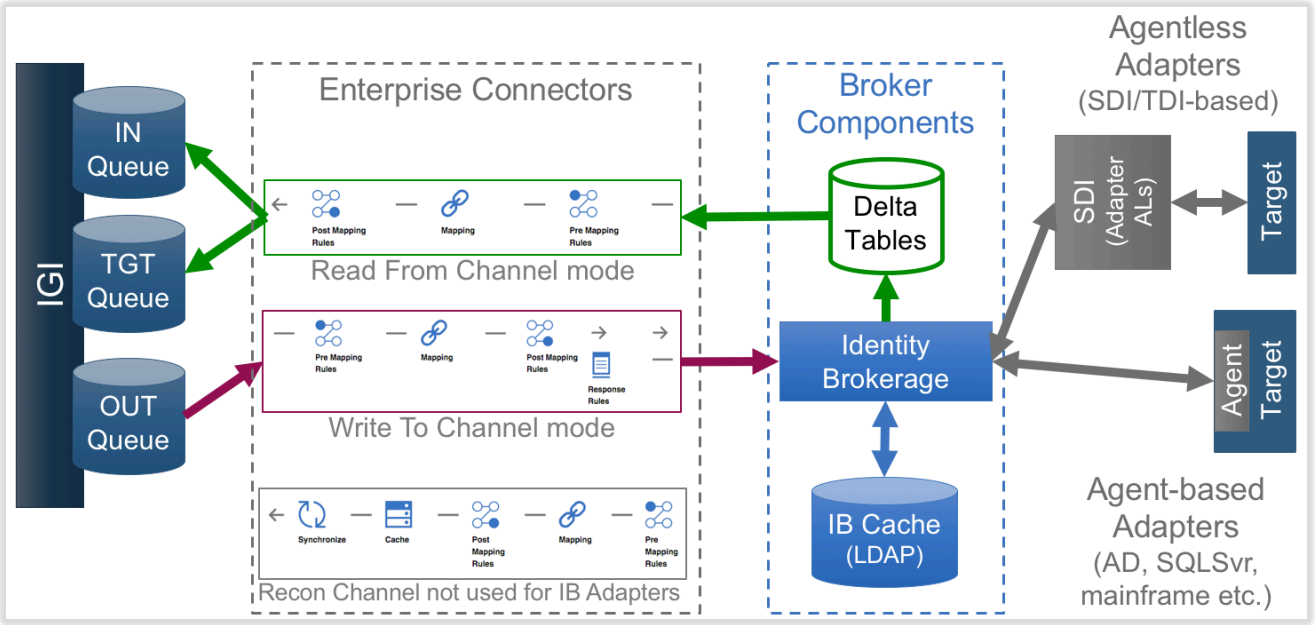
The modes are:

- **Read From Channel mode** – in this mode the Enterprise Connector framework is reading from the target system (via the connector or adapter logic). It may be processing User/OU data from a HR system, or account/permission data from a target system (account/access repository). This mode is used for both the legacy Enterprise Connectors and the Identity Brokerage Adapters.

- **Write To Channel mode** – in this mode the Enterprise Connector framework is reading from the OUT queue and pushing changes to a target system (via the connector or adapter logic). It will be processing account/permission data. This mode is used for both the legacy Enterprise Connectors and the Identity Brokerage Adapters.

- **Reconciliation Channel Mode** – this is similar to the Read From Channel Mode, but includes a Cacje and Synchronise mechanism. This is for target systems/connectors that cannot send deltas to IGI, so the framework needs to provide a mechanism to determine deltas. This mode is only used for legacy Enterprise Connectors as the Identity Brokerage has its own delta mechanism.

The following figure shows the Enterprise Connector framework with these three modes. This figure is focussed on Identity Brokerage Adapters as they are used far more often than legacy Enterprise Connectors.

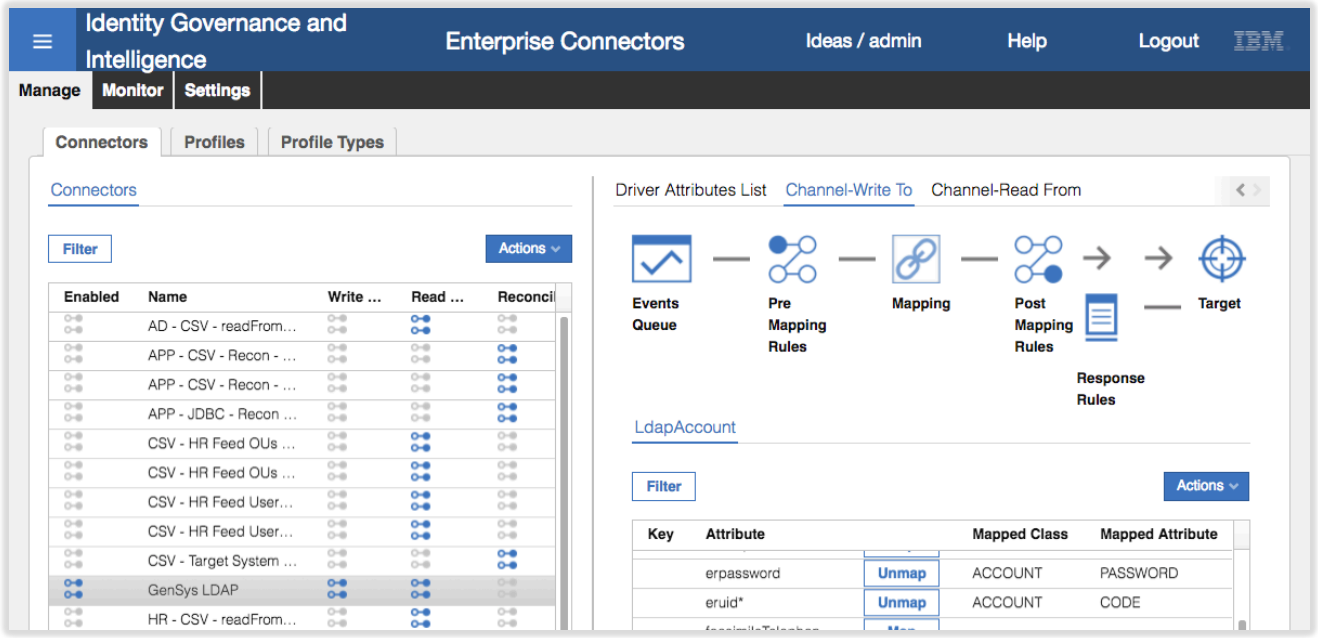


From a rules processing perspective, all rules processing occurs within the Enterprise Connector framework, so it doesn't matter whether it is a legacy Enterprise Connector or Identity Brokerage Adapter talking to the HR system or target system.

One of the main functions for the Enterprise Connector framework is data mapping between IGI and the Connectors/Adapters. There are three stages in any data mapping:

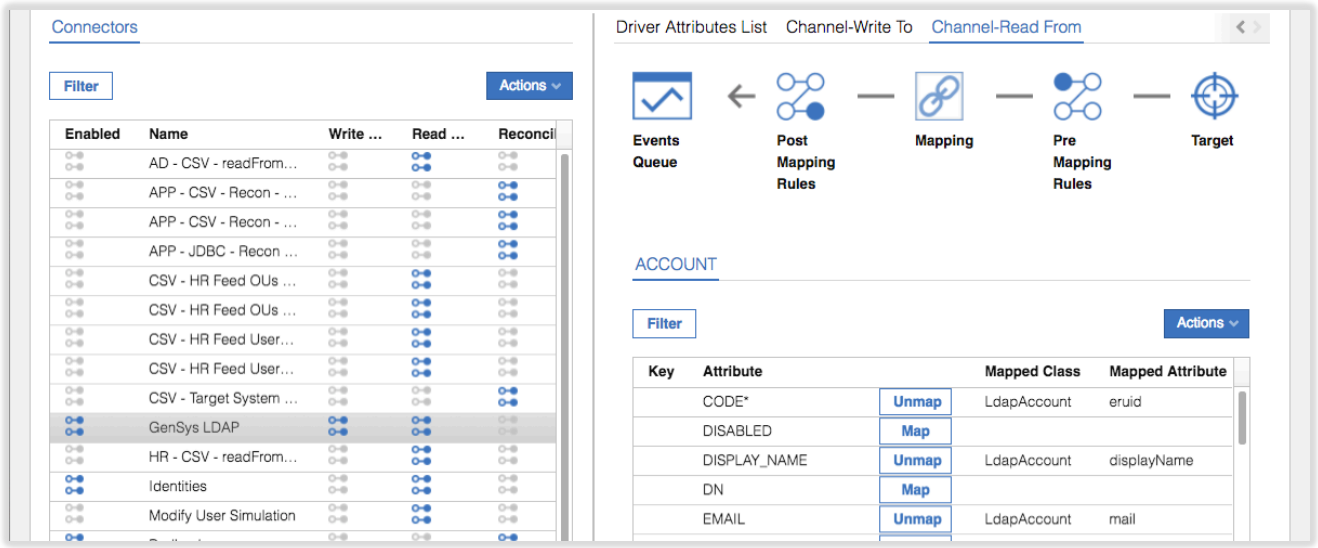
- Pre-mapping rules – rules applied to data before the mapping step is done,
- Mapping – the attribute to attribute mapping for the connector/adapter
- Post-mapping rules – rules applied to data after the mapping step is done.

The mapping step is an attribute-to-attribute mapping that is configured in the Enterprise Connectors module in the Admin Console. The following figure shows one of the Enterprise Connectors (which is actually an LDAP Brokerage Adapter) with part of the mapping view in the bottom right pane.



The figure shows that for the Channel-Write To mode, there are multiple Adapter attributes mapped to the IGI ACCOUNT attributes, such as adapter password (erpassword) is mapped to the ACCOUNT PASSWORD attribute, and the account id (eruid) is mapped to the ACCOUNT CODE attribute.

The Channel-Read From mapping for this connector is similar.



In this case the IGI account attributes are mapped to LdapAccount attributes, like IGI ACCOUNT CODE to eruid, DISPLAY_NAME to LdapAccount displayName and EMAIL to LdapAccount mail.

This visual mapping editor allows attributes to be mapped to;

1. Attributes (as shown above)
2. Constant values
3. Custom variables via a custom variable name

However there are no programmatic mechanisms available in the visual mapping editor. This is where the rules come in. The following sections will look at how we can apply rules to Read-From and Write-To connector modes. The Reconciliation mode is the same as the Read-From mode with respect to rules.

More information on Channel Modes can be found in https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/ECONN/Connector_Channel_Mode.html.

4.3 Rules within Connectors

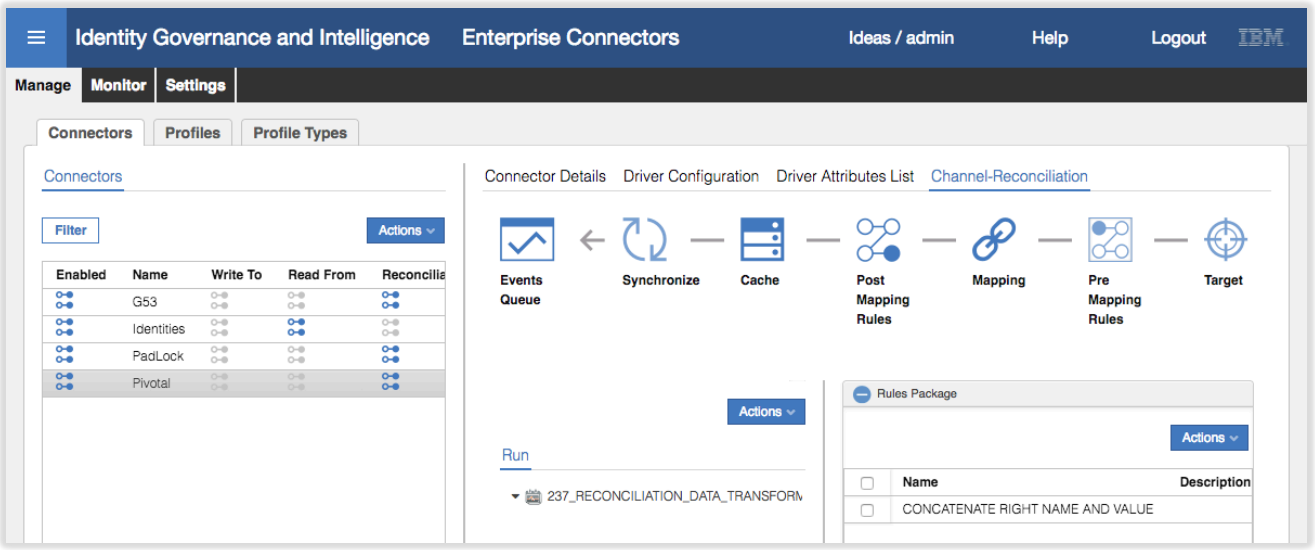
Rules within connectors are managed in the same way, irrespective of the rule function.

4.3.1 Rule Editing and Management

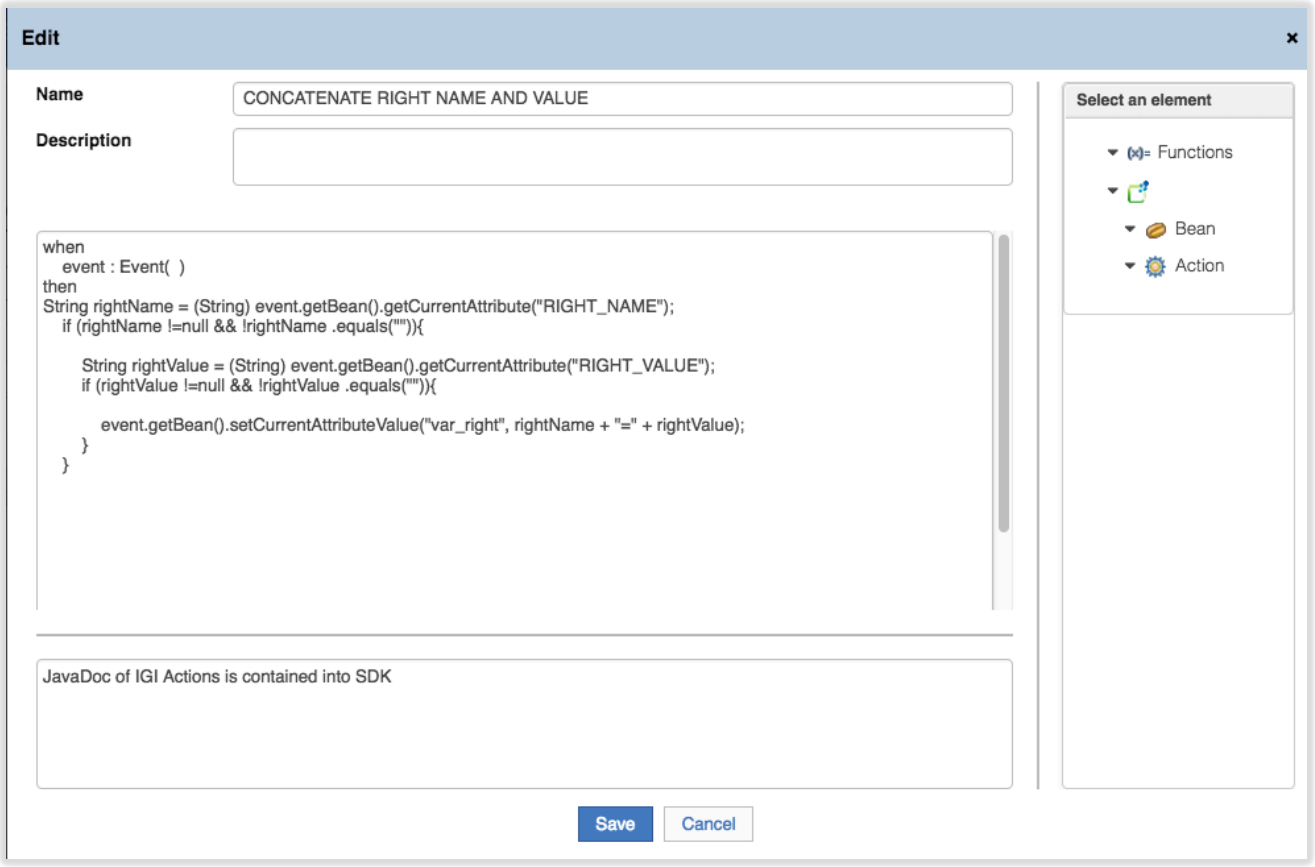
Rules for Enterprise Connectors are managed differently to other rules in IGI. They are managed from within the Enterprise Connector module, and a unique rule is associated with each function (pre-mapping, post-mapping and response) for each connector.

Rules may be built in the relevant connector or imported from another IGI module (Access Governance Core or Process Designer). There is no concept of rule class or rule sequences.

The following figure shows the rules interface for a pre-mapping rule in the Reconciliation Channel for an application connector.



The rules management interface is similar to what we saw with the event-based rules, with a flow ("Run") and a **Rules Package** and **Package Imports** section. Selecting **Actions > Modify** for a rule in the Rules Package presents the rule editor window.

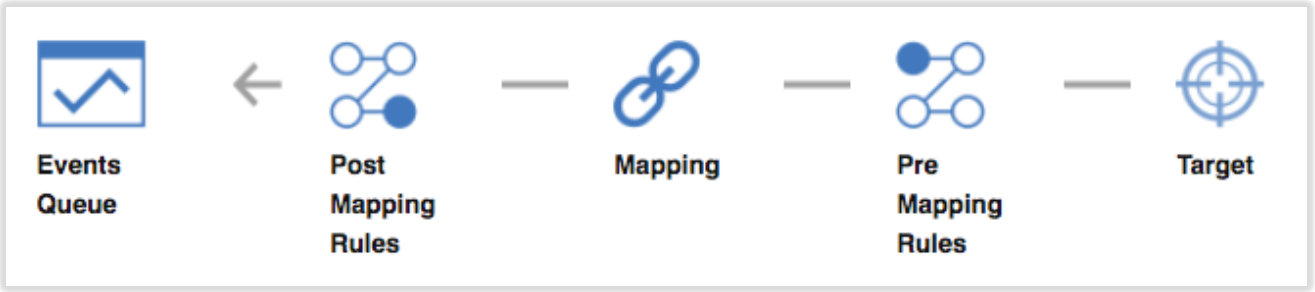


Selecting **Actions > Add** for a rule selected in the Rules Package will add it to the current flow. Selecting a rule in the flow and **Actions > Remove**, removes it.

Rules that have been exported from AGC or Process Designer can also be imported here using the **Actions > Run** option above the flow definition ("Run").

4.3.2 Rules for Read-From and Reconciliation Channel Mode in Connectors

Connectors with a Read-From Channel Mode are processing data flowing from the target, via the connector/adapter, the Enterprise Connector famework and into the IGI IN or TARGET queues.

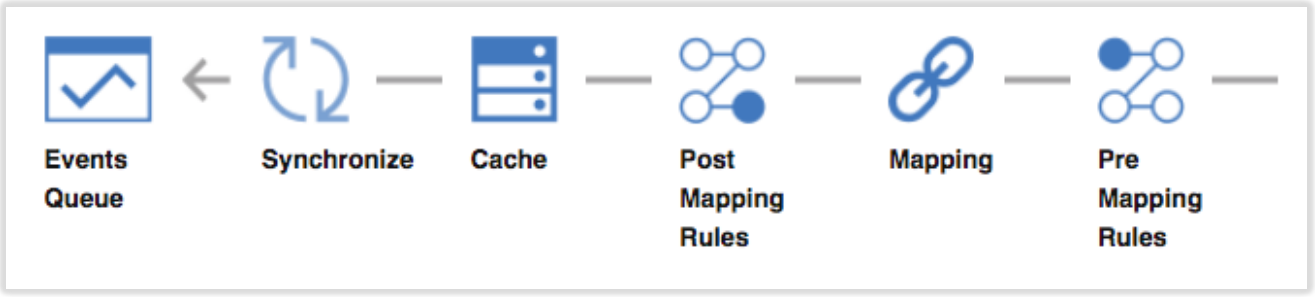


The Read-From channel has two rules:

- **Pre-Mapping Rules** – which processes data from the connector/adapter before it has been mapped to the corresponding IGI data. Thus they are focussed on the connector/adapter attributes.

- **Post-Mapping Rules** – which processes data after it has been mapped to the corresponding IGI data. Thus they are focussed on the IGI attributes.

Connectors with a Reconciliation Channel Mode are similar to Read-From Channel mode connectors, but process complete datasets and use a cache and synchronization process to generate deltas.

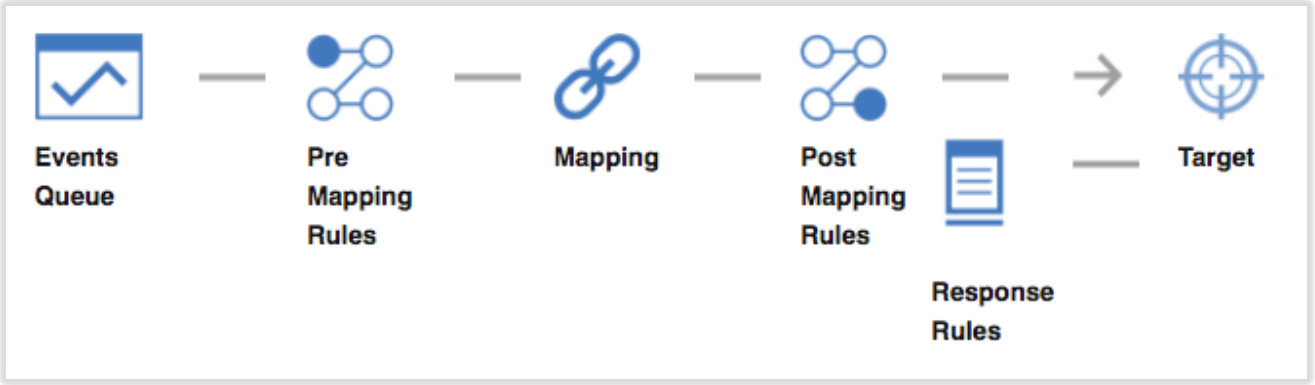


From a rules perspective, Reconciliation Mode connectors also have pre- and post-mapping rules.

The rules are driven by the Event() bean. There are no other beans in the working memory, however the rule may be able to access other IGI objects.

4.3.3 Rules for Write-To Channel Mode in Connectors

Connectors with a Write-To Channel Mode are processing data flowing from IGI (the OUT queue), through the Enterprise Connector framework, to the connector/adapter and on to the target.



The Write-To channel has three rules:

- **Pre-Mapping Rules** – which processes data from IGI before it has been mapped to the corresponding connector/adapter data. Thus they are focussed on the IGI attributes.
- **Post-Mapping Rules** – which processes data after it has been mapped to the corresponding connector/adapter data. Thus they are focussed on the adapter/connector attributes.
- **Response Rules** – these rules will write the response from the adapter/connector back to the event in the USER_EVENT_ERC table. This may be a success status, or some failure status with an error code.

The rules are driven by the Event() bean. There are no other beans in the working memory, however the rule may be able to access other IGI objects.

4.3.4 Choosing Between Pre-Mapping vs. Post-Mapping Rules

You may need to decide whether to apply your custom logic in a pre-mapping or post-mapping rule, but not sure which. This will come down to what data you can access from within the rule. After mapping, you will only have access to attributes that have been mapped. For example, if you want to access additional user attributes for an account on a provisioning activity, you will not have the id of the user object post mapping so you would need to do any lookups in the pre-mapping rule.

The following sections show some examples of this. These examples are from the Knowledge Center (5.2.3.1): https://www.ibm.com/support/knowledgecenter/SSGHJR_5.2.3.1/com.ibm.igi.doc/CrossIdeas_Topics/ECONN/prepost_mapping_rule_examples.html.

4.3.5 Example: Get User Attributes Outside of Event

In this example, there are some target system attributes that need to be mapped to IGI person object attributes but are not available in the Event bean. The code will use the user identifier in the Event bean to find the user object and pull additional attribute values.

This user data must be retrieved before the data mapping takes place (i.e. in the pre-mapping rules), and must be stored in the event attributes to be available to the post-mapping rules that follow. This is because the user's internal ID in USER_ERC is not one of the attributes that are mapped and is therefore discarded after the mapping takes place.

```
when
  event: Event()
then
//
          //String userId = (String) event.getBean().getCurrentAttribute("CODE");
          Long userErcId = (Long) event.getBean().getCurrentAttribute("USER_ERC_ID");

          ArrayList attributes = new ArrayList();
          attributes.add("GIVEN_NAME");
          attributes.add("SURNAME");
          attributes.add("EMAIL");
```

The first bit of code extracts the user's internal ID from the event bean (`event.getBean().getCurrentAttribute();`).

The next block creates a new array list and adds three string values to it, the three attributes to be mapped (GIVEN_NAME, SURNAME, and EMAIL).

```
          DataBean queryBackFilter = new DataBean("USER");
          queryBackFilter.setCurrentAttributeValue("ID", userErcId);
          //queryBackFilter.setCurrentAttributeValue("PM_CODE", userId);

          DataBean res = srcDriver.readObject(queryBackFilter, attributes);
              if (res == null) {
                  throw new Exception("User  not found");
              }
```

The next block creates a new DataBean (type "USER") and sets userErcId (the user from the Event bean) as the ID value. It then uses this to read the user object for this user and the three attributes in the attributes array.

```
        Object GIVEN_NAME = res.getCurrentAttribute("GIVEN_NAME");
        Object SURNAME = res.getCurrentAttribute("SURNAME");
        Object EMAIL = res.getCurrentAttribute("EMAIL");

        event.setEventAttribute("GIVEN_NAME", GIVEN_NAME);
        event.setEventAttribute("SURNAME", SURNAME);
        event.setEventAttribute("EMAIL", EMAIL);
```

The final block pulls the attributes from the results of the search and sets them to the matching attributes on the Event bean. These would then be mapped to corresponding adapter/connector attributes and passed to the adapter/connector.

4.3.6 Example: Set a Random Password on Re-Enabled Account

This rule generates an eight-character password for the restored LDAP account. The last four digits are random. The calculated value is added to the data bean in a current attribute that is named erLdapPwdReset. This name is expected by the LDAP adapter.

The password attribute does not need to be mapped as it is created by the rule at post-mapping time.

The rule code is:

```
when
  event: Event()
then
//
        if (event.getOperation() == EventOperationType.OPERATION_ACCOUNT_ENABLE) {

             Random rnd = new Random(System.currentTimeMillis());
             int n = 1000 + rnd.nextInt(9000);
             String randomPwd = "Ibm$" + n;
             event.getBean().setCurrentAttributeValue("erpassword", randomPwd);
        }
```

The code checks if this is an account enable operation.

These Event() constants aren't documented yet – you won't find the Event() bean in the 5.2.3.1 SDK.

If it is an account enable operation, it will generate a random four-digit number and prepend "Ibm$" to make the random password. This password is then set on the erpassword attribute on the event bean.

The erpassword attribute is common to all Identity Brokerage Adapters (if you were using a legacy Enterprise Connector, you would need to specify the appropriate account password attribute).

To run this rule, you must import additional Java packages. Select the Package Imports accordion pane and write the following lines:

```
import com.crossideas.ideasconnector.common.enums.EventOperationType
import java.util.Random
```

> This code may be able to leverage the password generation mechanism in IGI. It would need to find the PwdConfig (i.e. the account configuration) for this account/application then generate a password that complies with the relevant password strength rules. See Example: Set Random Password for Ideas Account on New User on page 36 for an example of setting the Ideas account password.

4.3.7 Example: Email New Password to User

This rule follows the previous two. The rule sends an email with the generated password to the user of the restored account. The email address is retrieved from the session attributes that were retrieved by the pre-mapping rule at the top of this section.

> Before you run the rule, you must have defined the Password Reset template email template that automatically notifies the user of the new password. Make sure that the placeholders that you use in the template correspond to the elements of the data map; that is, $P{givenName}, $P{surname}, and $P{password}.

The code is:

```
when
  event: Event()
then
//
 //----------- CONFIGURATION ---------------
 String TEMPLATE_NAME = "Password Reset template";
 String LANG = "EN";
 String MAIL_FROM = "igi@mycompany.com";
 String SYSTEM_NAME = "MyLDAP";
 //-----------------------------------------

 String newPassword = (String) event.getBean().getCurrentAttribute("erLdapPwdReset");
```

The first bit of code sets variables;

- The TEMPLATE_NAME (email template, set in AGC > Configure > Notifications),
- The language,
- The sender of the email (system account),
- The system name (in this case MyLDAP), and
- The password set in the last rule

```
 if (newPassword != null) {

  // get data from the event. A pre-mapping rule must fill these data.
  String identityEmail = (String) event.getEventAttribute("EMAIL");
  String givenName = (String) event.getEventAttribute("GIVEN_NAME");
  String surname = (String) event.getEventAttribute("SURNAME");

    if (identityEmail != null) {

     ArrayList<String> recipients = new ArrayList<String>();
     recipients.add(identityEmail);

     Map<String, String> dataMap = new HashMap<String, String>();

     dataMap.put("password", newPassword);
     dataMap.put("givenName", givenName);
     dataMap.put("surname", surname);
     dataMap.put("system", SYSTEM_NAME);

     EmailDataBean emailBean = new EmailDataBean(MAIL_FROM, recipients);
```

```
      WebEmailAction.submitEmail(sql, dataMap, "", "Connector", LANG, "",
  IdeasApplications.EMAILSERVICE.getName(), TEMPLATE_NAME, emailBean);
                                    }
                                }
```

The part of code will only run if there's a password defined (which it should if the last rule ran successfully).

It will retrieve the three attributes (two names and email) set in the first rule (from the person object). If the email is defined (i.e. not null) it creates a new HashMap and poplulates it with the new password, given name, surname and system name (in this case "MyLDAP", but it could probably be extracted from the event attributes also).

Finally it creates a new EmailDataBean with the predefined MAIL_FROM (igi@mycompany.com) and the recipients list (i.e. the email from the user object and set on the event object in the first rule).

The `WebEmailAction.submitEmail` action will submit the email. The mail template (TEMPLATE_NAME = "Password reset template") contains variable fields ($P{givenName}, $P{surname}, and $P{password}) that the hashmap values will replace when the email is rendered and sent.

To run this rule, you must import additional Java packages. Select the Package Imports accordion pane and write the following lines:

```
import com.crossideas.email.common.action.WebEmailAction
import com.crossideas.email.common.bean.EmailDataBean
import com.engiweb.toolkit.common.enums.IdeasApplications
```

You must also add a global variable. Select the Package Imports accordion pane and write the following line:

```
global com.engiweb.pm.dao.db.DAO sql
```

4.3.8 Example: Create Custom Attributes and Use in Data Mapping Rules

The following example is similar to the first data mapping example above, but adds in creation of a custom variable.

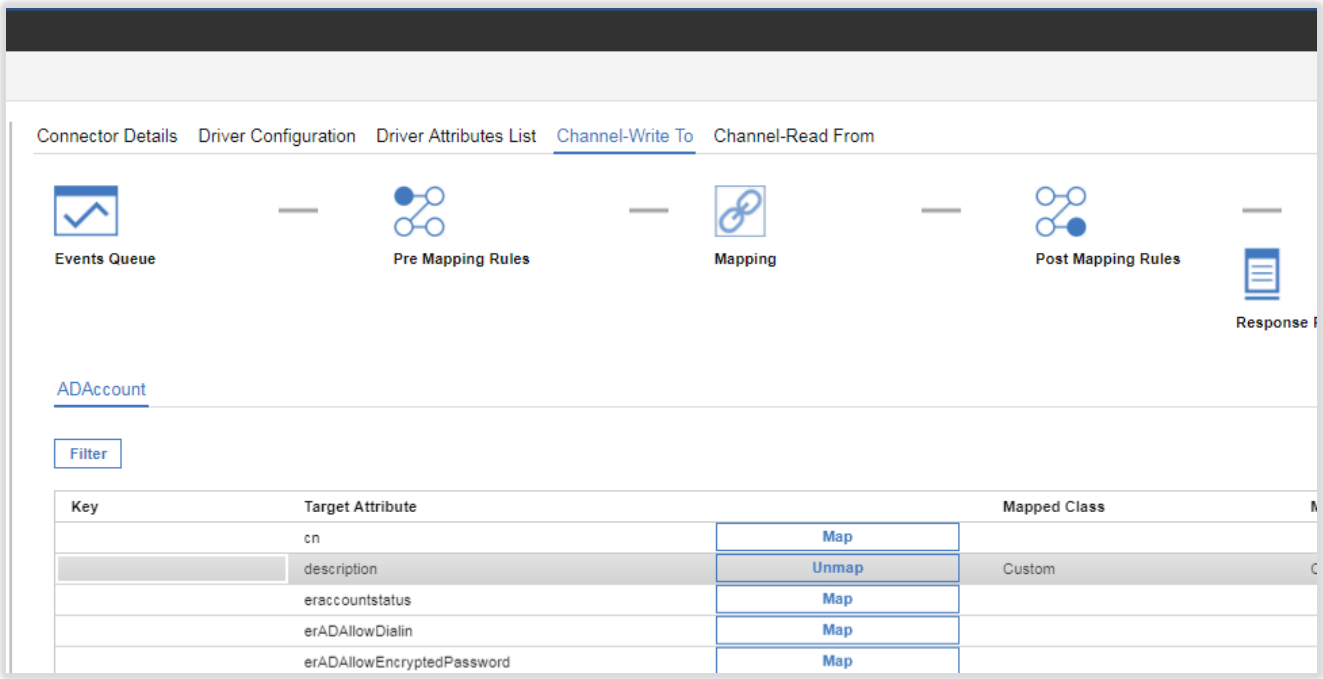
The code is:

```
when
    event : Event(  )
then
//
//String userId = (String) event.getBean().getCurrentAttribute("CODE");
Long userErcId = (Long) event.getBean().getCurrentAttribute("USER_ERC_ID");

ArrayList attributes = new ArrayList();
attributes.add("GIVEN_NAME");
attributes.add("SURNAME");
attributes.add("EMAIL");

DataBean queryBackFilter = new DataBean("USER");
queryBackFilter.setCurrentAttributeValue("ID", userErcId);
```

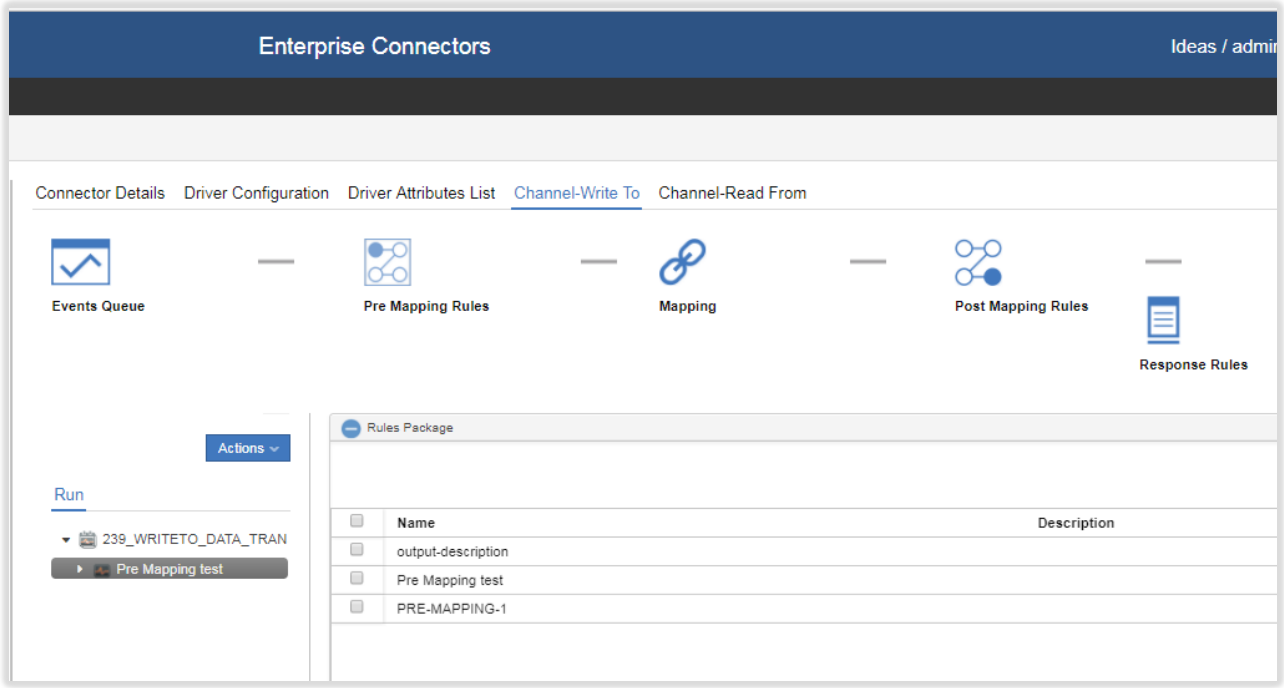
```
//queryBackFilter.setCurrentAttributeValue("PM_CODE", userId);

DataBean res = srcDriver.readObject(queryBackFilter, attributes);
 if (res == null) {
     throw new Exception("User  not found");
 }

Object GIVEN_NAME = res.getCurrentAttribute("GIVEN_NAME");
Object SURNAME = res.getCurrentAttribute("SURNAME");
Object EMAIL = res.getCurrentAttribute("EMAIL");

event.setEventAttribute("GIVEN_NAME", GIVEN_NAME);
event.setEventAttribute("SURNAME", SURNAME);
event.setEventAttribute("EMAIL", EMAIL);
DataBean eventBean = event.getBean();

eventBean.setCurrentAttributeValue("CUSTOM_DESC", "Test Description" + " - " + GIVEN_NAME
+ " - " + SURNAME + " - " + EMAIL);
```

The change from the earlier example is highlighted. It is setting a custom attribute to a combination of strings and the GIVEN_NAME, SURNAME and EMAIL. The custom attribute name is CUSTOM_DESC.

The custom attribute was defined in the Mapping step of the Enterprise Connector.



It was added as a new Pre-Mapping rule.



This figure shows the rule, Pre-Mapping test, being applied to the pre-mapping rule from the Rules Package, similar to how it's done from the normal rules interface.

4.3.9 Example: Date Manipulation in a Pre Mapping Rule

This example was provided by Vaughan Harper and was used with a CSV Connector.

It formats a date coming from the CSV file into the format needed by IGI, and is based on one of the rules in the documentation (was in 5.2.3 but removed in 5.2.3.1 docs).

The Package Header for this rule must include:

```
import java.text.SimpleDateFormat
import java.util.Date
import java.sql.Timestamp
```

The rule itself is:

```
when
    event : Event(  )
then
//Pre Mapping Rule to parse DATE_OF_BIRTH
//based on example of a Pre Mapping Rule parse DATE_OF_BIRTH in IGI 5.2.2 Admin Guide
    log.debug("Entered DATE_OF_BIRTH pre-mapping rule");
    String attributeName = "DATE_OF_BIRTH";
    DataBean dBean = event.getBean();
    String dateString = (String) dBean.getCurrentAttribute(attributeName);
    log.debug("DATE_OF_BIRTH pre-mapping rule: dateString: " + dateString);
```

```
    if (dateString != null) {
      SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
```

```
      try {
        Date date = dateFormat.parse(dateString);
        Timestamp timestamp = new java.sql.Timestamp(date.getTime());
        dBean.setCurrentAttributeValue(attributeName, timestamp);
        log.debug("DATE_OF_BIRTH pre-mapping rule: updated attribute " + attributeName + "
with date");
      } catch (Exception e) {
        dBean.stripCurrentAttribute(attributeName);
        String errorMessage = "DATE_OF_BIRTH wrong format: " + dateString;
        String userid = (String) dBean.getCurrentAttribute("USERID");
        if (userid != null) {
          errorMessage = errorMessage + " for userid: " + userid;
        }
        log.error(errorMessage);
      }
    }
log.debug("Leaving DATE_OF_BIRTH pre-mapping rule");
```

More details on the rule development can be found in the rules examples folder.

This concludes the Enterprise Connector mapping rule examples, and the chapter on Enterprise Connector rules.

5 Rules for Other Operations in IGI

The last two chapters focussed on event-based rules; the rules used to enhance and extend processing of events in queues and in the Enterprise Connectors framework. This chapter covers all of the other uses of rules within IGI.

5.1 Introduction to Rules for Other Operations in IGI

Rules can be used to exend and enhance the provided IGI funtionality in many parts of the product. Recall the rules classes listed earlier in the document:

Rule Class	Module	Fixed Flow (Sequence)	Triggered by
Live Events	Access Governance Core	Yes	Events execution (with exception of creation events)
Deferred Events	Access Governance Core	Yes	Deferred Events execution (with exception of creation events)
Authorization Digest	Access Governance Core	No	Fulfilment of a certification campaign
Advanced	Access Governance Core	No	Not triggered but can be scheduled by a job of Task Planner
Account	Access Governance Core	No	Account creation
Password	Access Governance Core	No	Password generation/change
Attestation	Access Governance Core	No	Creation of a data set for a certification campaign
Hierarchy	Access Governance Core	No	Building of an attribute hierarchy
Workflow	Process Designer	No	Pre-action or post action that is related to a workflow activity
Advanced	Access Risk Controls for SAP	Yes	SAP system operation

The previous chapter looked at the event-based rules; the Live Events rules, the Deferred Events rules, and the rules used in the Enterprise Connectors module.

We can also have rules for:

- **Certification Campaigns** to:
  - Drive fulfillment of certification campaigns (Authorization Digest rules), such as custom behaviour when an access is revoked.
  - Build certification datasets for continuous campaigns (Attestation rules)

- Other **Access Governance Core** functions to:
  - Perform custom account management activities (Account rules), such as userid creation
  - Extend password strength policy (Password rules), and
  - Build custom attribute hierarchies (Hierarchy rules)

- **Task Planner** to:
  - Run custom activity as a background task on a scheduler (Advanced rules)

- **Process Designer** to:
  - Run pre- and post-actions on activities (Workflow rules) to alter the behaviour of workflow processes

These will be covered in later sections in this chapter.

We can also have Advanced rules for the Access Risk Control for SAP module. However this is a very specialized area and not covered in this document.

Rules are maintined in the rules interface (**Configure > Rules**) in the **Process Designer** module (for Workflow class rules) and **Access Governance Core** (for all other rules). The interface is as we have seen.

5.2 Rules in Campaigns

There are two classes of rules involved in certification campaigns; Attestation rules to populate the certification datasets of continuous campaigns and Authorization Digest rules to drive activity resulting from a campaign review.

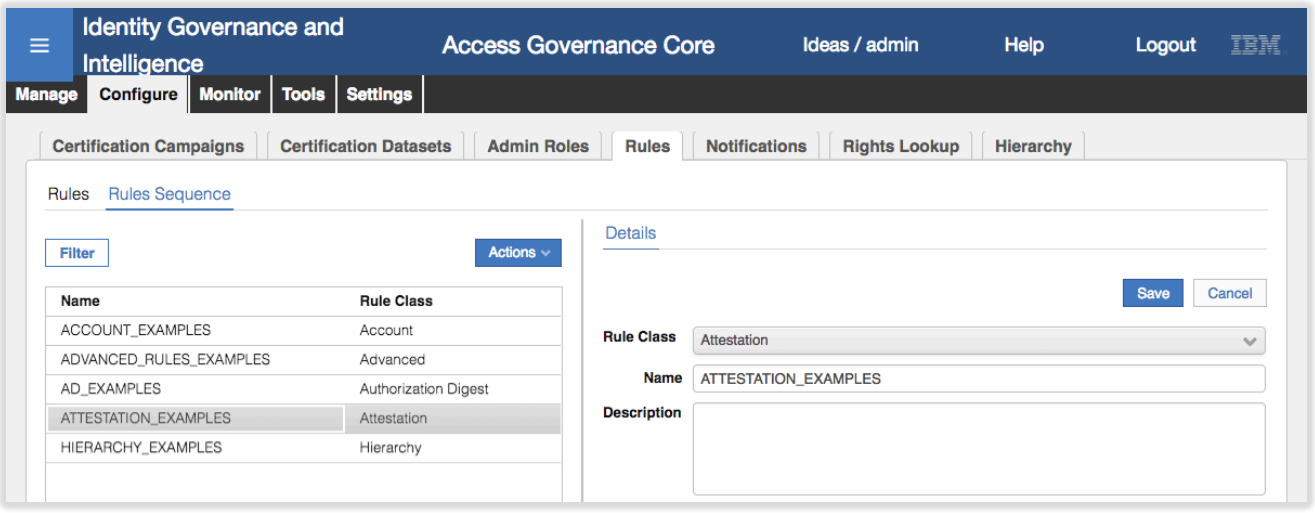
5.2.1 Rules for Populating Campaign Datasets

Certification campaigns use certification datasets to define the content of the campaign (e.g. user entitltmeents to be reviewed).

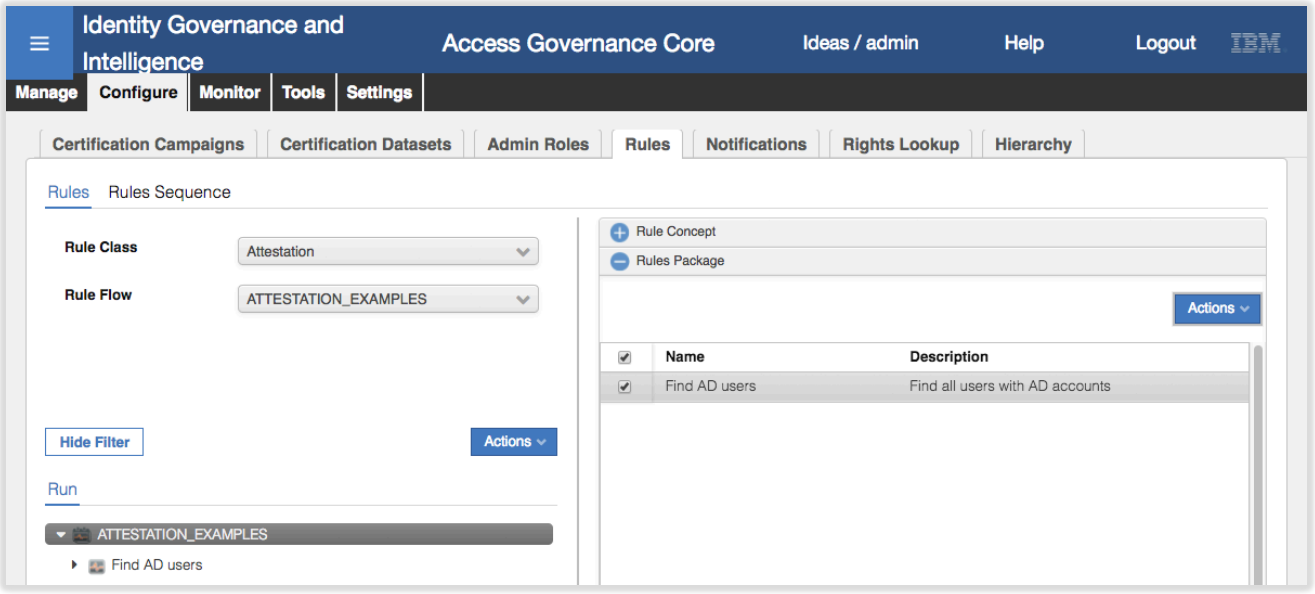
Populating Dataset at Campaign Launch with Attestation Rules

Campaign dataset content (users) can be defined at campaign launch time by using **Attestation** rules. Rather than having a static definition of users in the white-lists and black-lists, we can run a rule that will build the content.

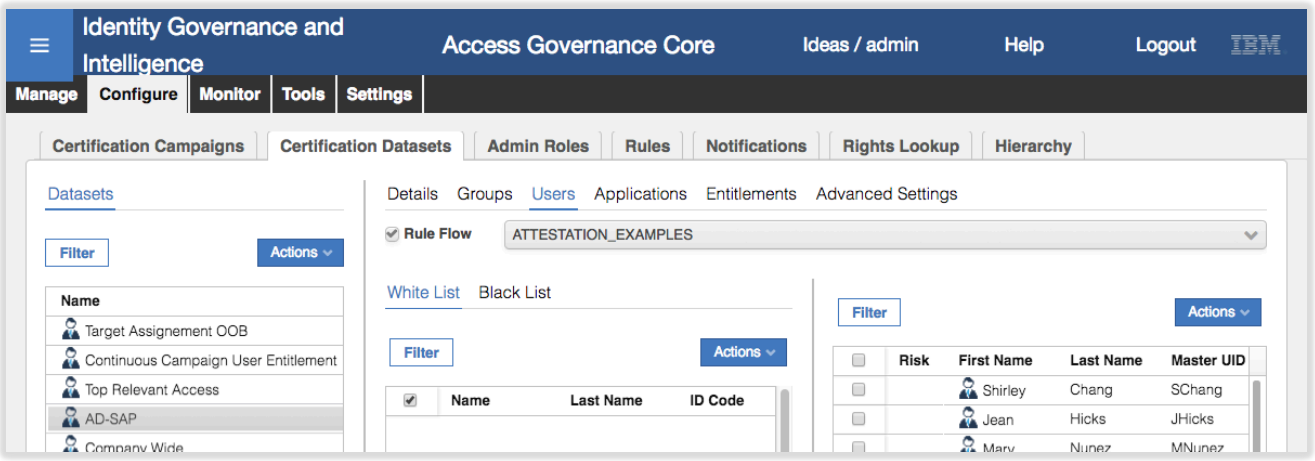
This approach requires a Rules Sequence of type Attestation (**AGC > Configure > Rules > Rules Sequence**).



This rules sequence must have a Rule Flow defined, with one or more rules (**AGC > Configure > Rules > Rules**).



This Rules Sequence Flow is then attached to the Certification Dataset (**AGC > Configure > Certification Datasets > dataset > Users**, enable Rule Flow and select the appropriate sequence flow).



If you want to have different rules for different campaign datsets, you must have different rules sequences (of type Attestation), each containing the different rules in the flow.

Dynamically Populating Continuous Campaign Datasets with Event-based Rules

Campaigns can also be set to run continuously, with the contents of the campaign fed into the campaign dataset (an internal IGI data structure, not an external file) by rules. We saw examples of this in the last chapter: Example: User Move Triggers Continuous Certification Campaign (on page 38), Example: New Permission Assignments Drive a Continuous Campaign (on page 57), and Example: New Permission Mapping Drives Continuous Campaign (on page 63).

5.2.2 Example: Attestation Rule - TBA

To be added.

5.2.3 Authorization Digest Rules for Post-Campaign Activity

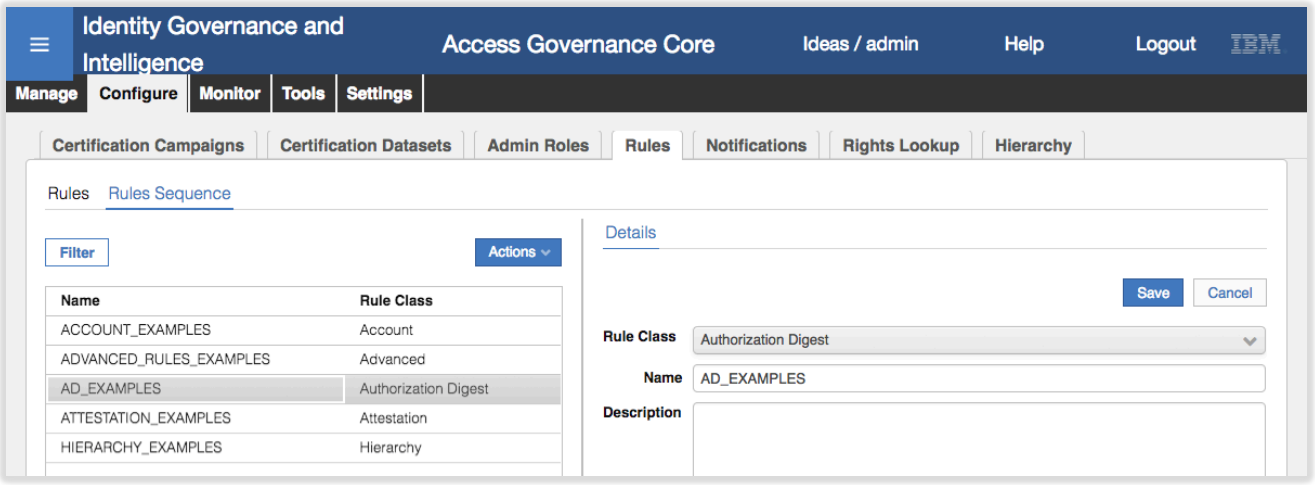
Whilst the sections above describe using rules to populate the contents of the campaign dataset driving a campaign, the **Authorization Digest** rules can drive post-review activity when a reviewer has reviewed access.

The Certitification Campaign fulfillment configuration (AGC > Configure > Certification Campaigns > campaign > Fulfillment) defines what IGI does in response to a reviewer revoking access. It can:

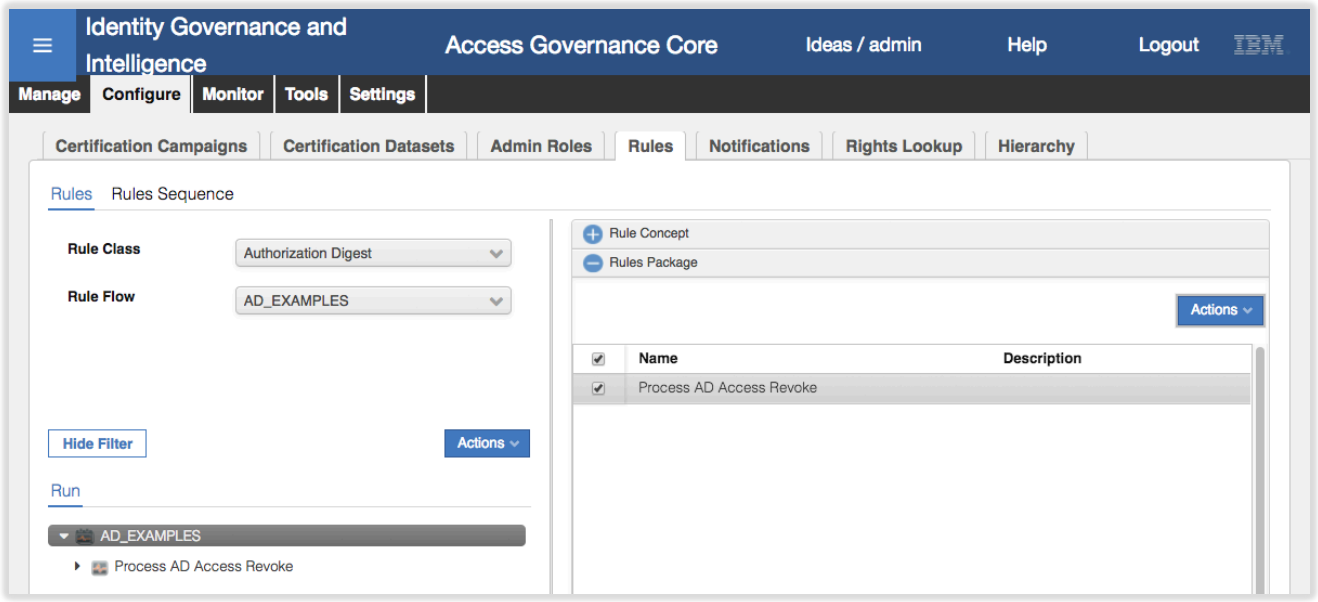
- Not delete the user entitlement, just mark in the history that it was flagged for revoke ("Logical deletion (just flagged out").
- Send a provisioning event to remove the permission from the user account on the target system ("Physical deletion"), with an optional grae period.
- Run a rule ("Custom Behaviour").
- Trigger a workflow process ("Physical deletion after workflow"), where the process GEN activity is replaced by the campaign execution, so the AUTH (and optionally the EXE) steps are run in response to a remove permission change. This has the same outcome as if someone had requested an access be removed in the Access Request Management module (Service Center). Different workflow processes can be set for non-Admin Roles and Admin Roles.

The ability to run a workflow process on an access revoke meets many customer requirements (e.g. "I want the end user/department head/application owner to be able to review/stop and access revoked by a manager").

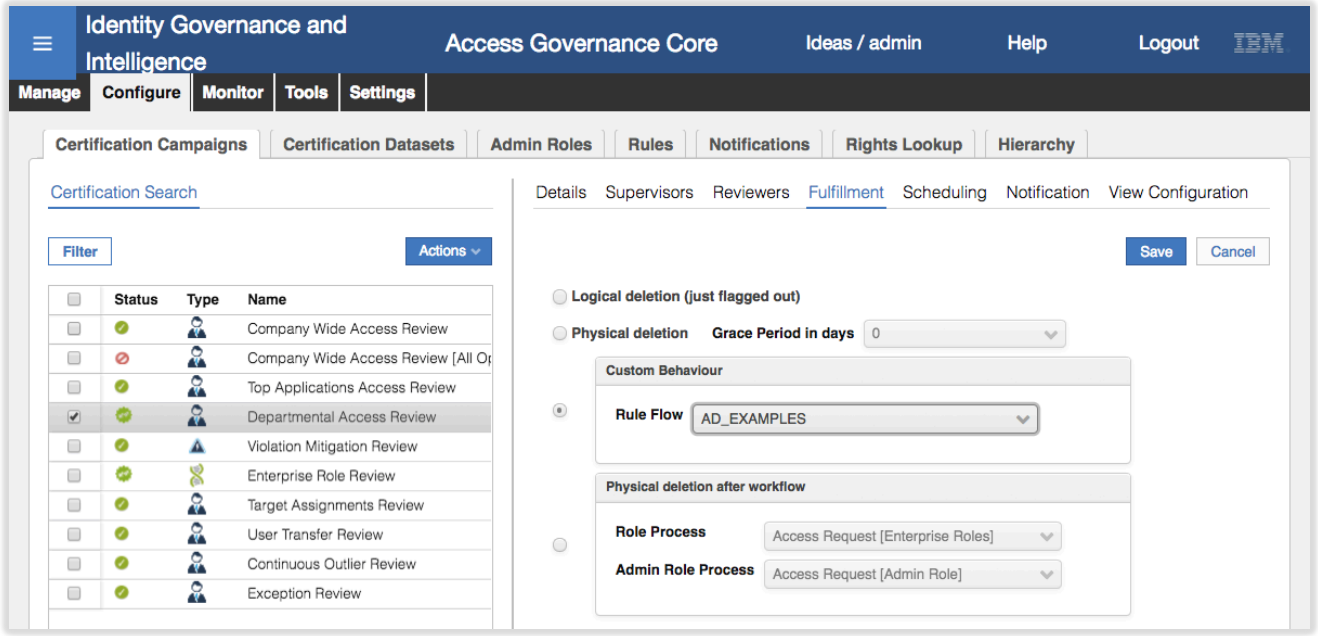
However there are still scenarios where being able to run a rule is needed. For example, customers sometimes want to chain reviews; manager reviews, then application owner reviews. Rules can be written to populate a certification dataset based on the results of another campaign.
This approach requires a Rules Sequence of type Authorization Digest (**AGC > Configure > Rules > Rules Sequence**).



This rules sequence must have a Rule Flow defined, with one or more rules (**AGC > Configure > Rules > Rules**).



This Rules Sequence Flow is then attached to the Certification Camapign (**AGC > Configure > Certification Camapigns > campaign > Fulfillment**, select **Custom Behaviour** and select the appropriate sequence flow).



If you want to have different rules for different campaigns, you must have different rules sequences (of type Authorization Digest), each containing the different rules in the flow.

Authorization Digest rules can also be run via a scheduled Task tied to an Advanced Rule Flow job. This would allow batch processing of entitlement revokes.

For an Authorization Digest rule you will have access to two beans in the working memory – the UserBean and the EntitlementBean.

5.2.4 Example: Authorization Digest Rule - TBA

To be added.

5.3 Other Rules in Access Governance Core

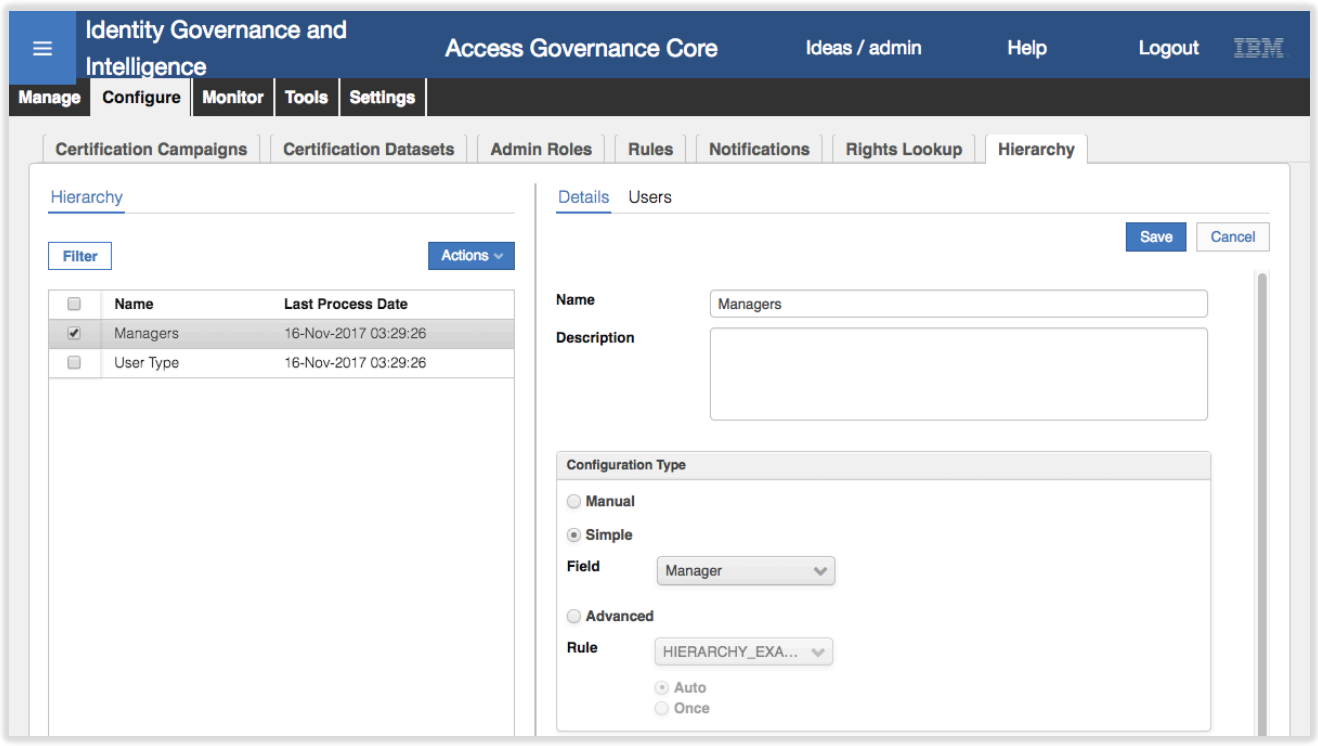
There are three other rule types that can be associated with object management in Access Governance Core:

1. Hierarchy rules allow complex attribute hierarchies to be defined
2. Account rules allow programmatic logic to be used to create account userids
3. Password rules allow extra password strenght checking for account configurations

These are discussed in the following sections

5.3.1 Hierarchy Rules

Attribute hierarchies represent groups of users based on attribute values, such as all users grouped by their manager (e.g. beloning to manager "Dfox") or by type (e.g. all users of type "Employee"). Most attribute hierarchies are based on a single user attribute (like manager or type).



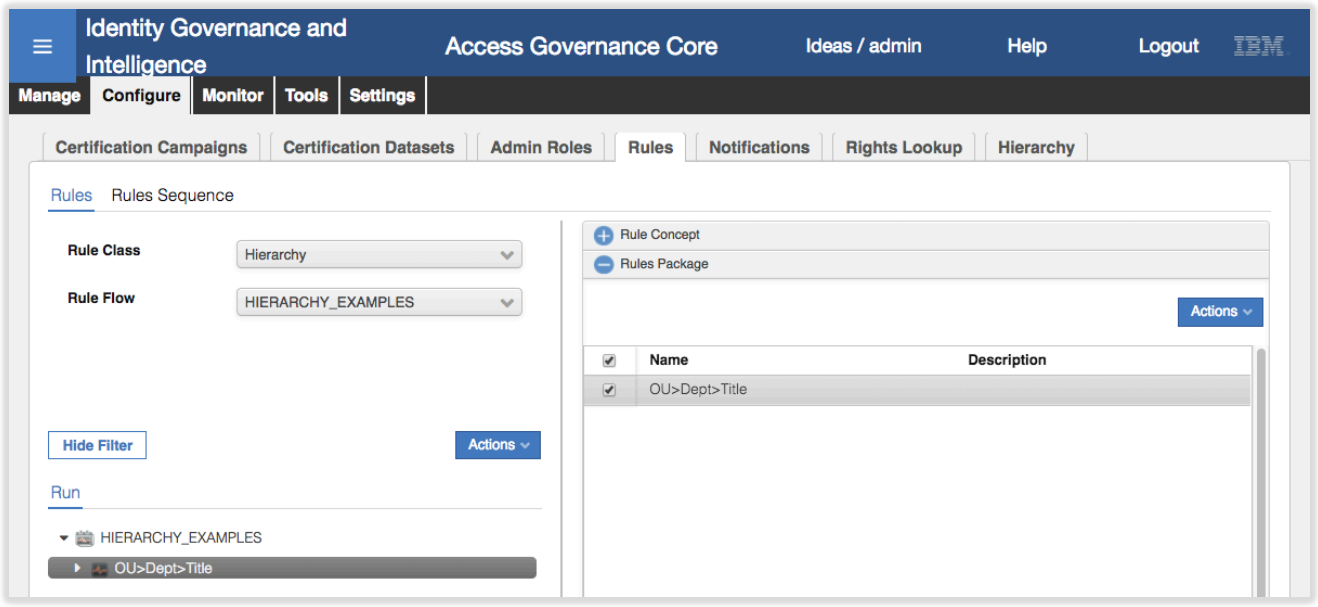
There are three ways an attribute hierarchy can be built (Configuration Type setting for a hierarchy):

- Manually (Manual),
- By tying the attribute groups to user attribute values (Simple), or
- By running a rule to build the hierarchy (Advanced)

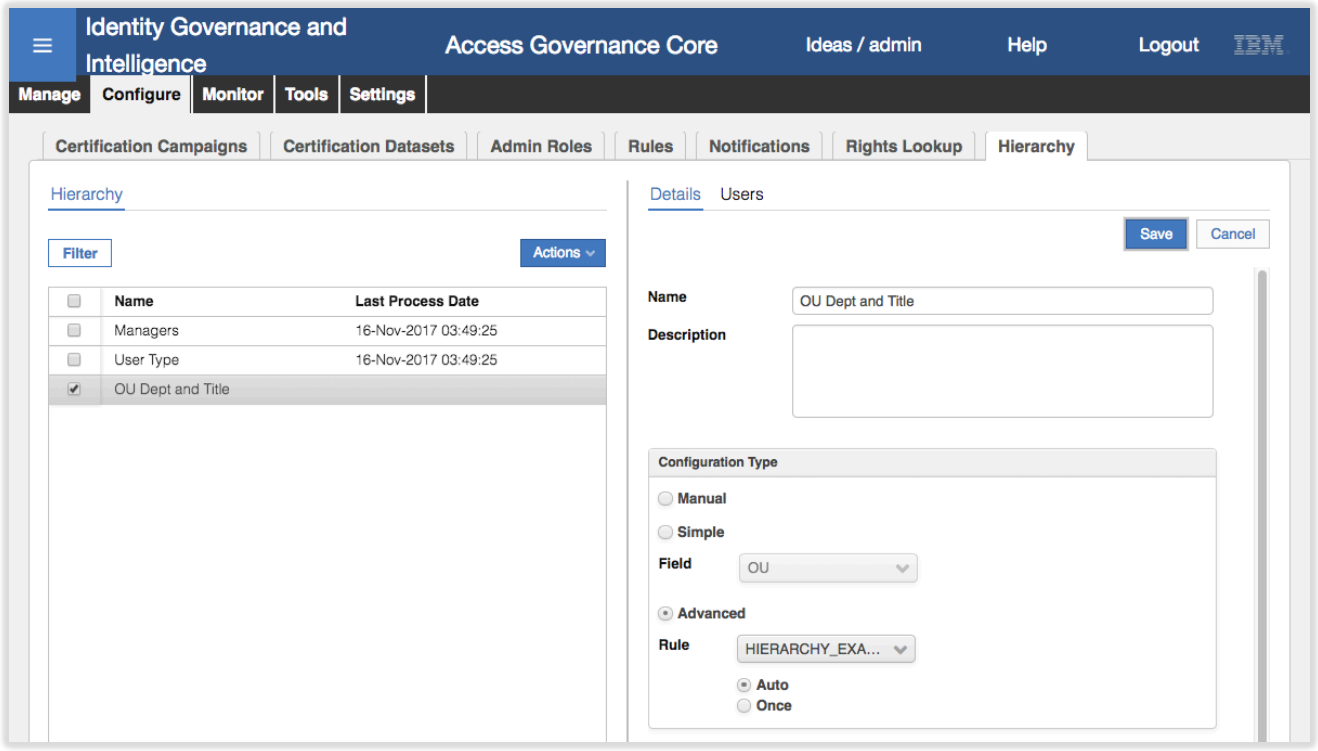
For Simple hierarchies, the hierarchy can be manually (re)built, or it can run on a schedule using a task with an AttributeHierarchyRefresh job. The default task for this is the Hierarchies Refresh task.

The Advanced hierarchies can use a scheduled task (as with the Simple ones) or only run once.

As with the other non-event rules, there must be a Rule Sequence of type Hierarchy defined, and this Rule Sequence must have one or more rules assigned to the Rule Flow (after creating them in the Rules Package).



This Rules Sequence Flow is then mapped to the relevant Hierarchy (AGC > Configure > Hierarchy) by setting the Advanced Configuration Type and selecting the rule sequence.



If you want to have different rules for different hierarchies, you must have different rules sequences (of type Hierarchy), each containing the different rules in the flow.

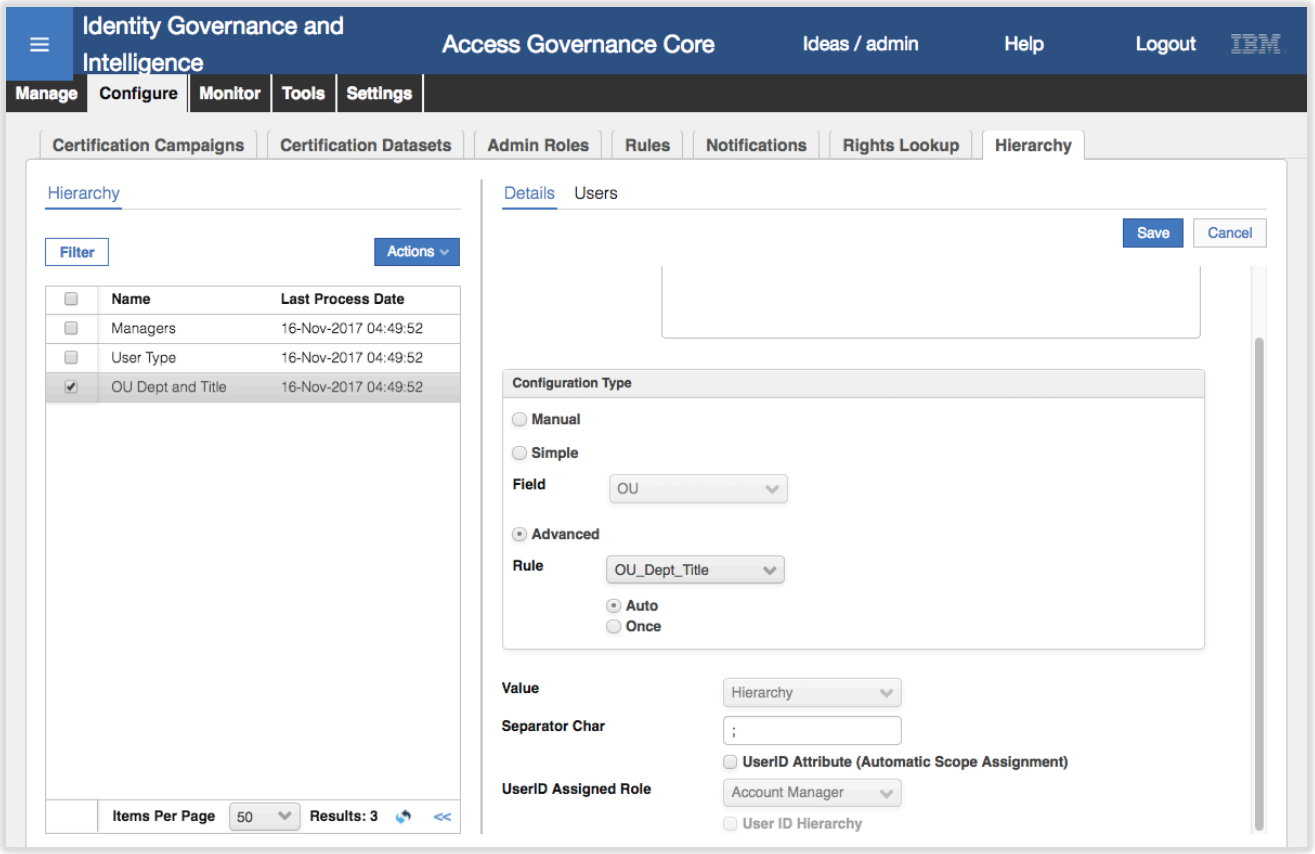
A hierarchy rule attached to a hierarchy will be run once for every user in IGI. If you have 10,000 users, it will run 10,000 times each rebuild (if scheduled). You should consider efficient coding and limited logging (you don't want to fill the VA disk with trivial log messages).

The working memory for hierarchy rules includes the UserBean, UserExtInfoBean and a ResultBean for each user. The input is the user and extended attributes for the user, the output is a string representing a hierarchy group for the user. IGI will populate the hierarchy based on all the result beans.

5.3.2 Example: Rule to Build a Hierarchy On OU, Department and Title

Often a reporting structure is more complex than just a HR-based org structure. This example builds a hierarchy based on OU (where the OUs represent geographies), department and user title.

The hierarchy is defined as shown in the following figure.



The configuration type is set to Advanced and the Rule is the custom rule flow ("OU_Dept_Title"). The value is set to Hierarchy with a separator character of ";". This matches how the rule writes out the output.

> It may be a bug in 5.2.3.1, but you cannot select the Value and Separator Char values once you select the Configuration Type as Advanced. If you need to set them, select a Simple Configuration Type, set them, save, then change to Advanced, select your Rule and save again.

The code for this rule is:

```
when
    userBean : UserBean(  )
    userInfoList : ArrayList(  )
    resultBean : ResultBean(  )
then
/*NATION for first, Attr3->DepartmentCode,Attr4->Title*/

String geo=null;
String department=null;
String title=null;
```

The first bit of code assigns the working memory objects; the UserBean, the extended user attributes passed in the ArrayList bean, and a ResultBean which will contain the built hierarchy group string.

It creates empty (null) variables for geo (geography/OU), department and title.

```
for (int i=0;i<userInfoList.size();i++){
  UserExtInfoBean extInfoBean = (UserExtInfoBean)userInfoList.get(i);

  String name = extInfoBean.getName();
  String value = extInfoBean.getValue();

  if(name.equalsIgnoreCase("OU")&&value!=null){
    geo = value;
  }

  if(name.equalsIgnoreCase("department")&&value!=null){
    department= value;
  }

  if(name.equalsIgnoreCase("Title")&&value!=null){
    title= value;
  }
}
```

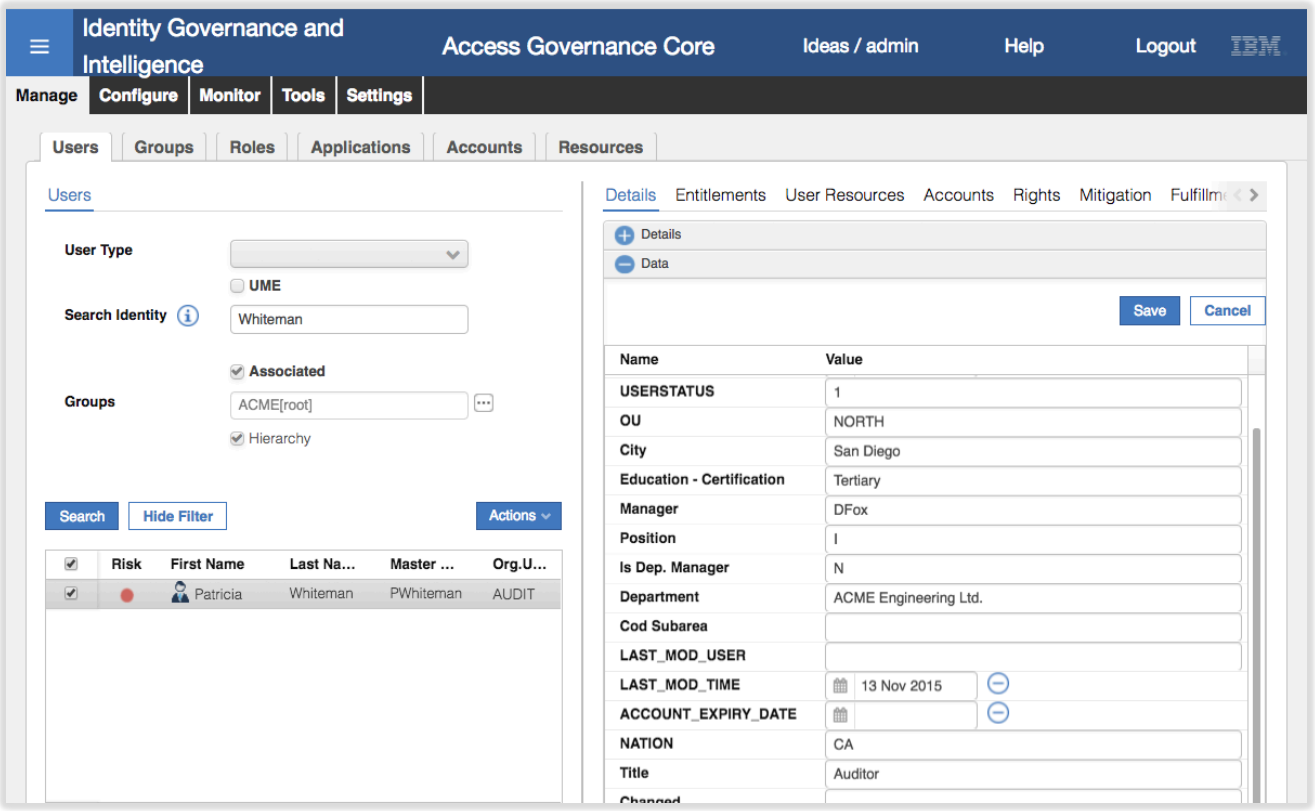
This part of the code will run through the entire attribute set for the user using the userInfoList ArrayList() bean. For each entry, it will parse out the attribute name and value from the associated UserExtInfoBean.

It assigns the geo, department and title local variables based on the OU, department and title attributes.

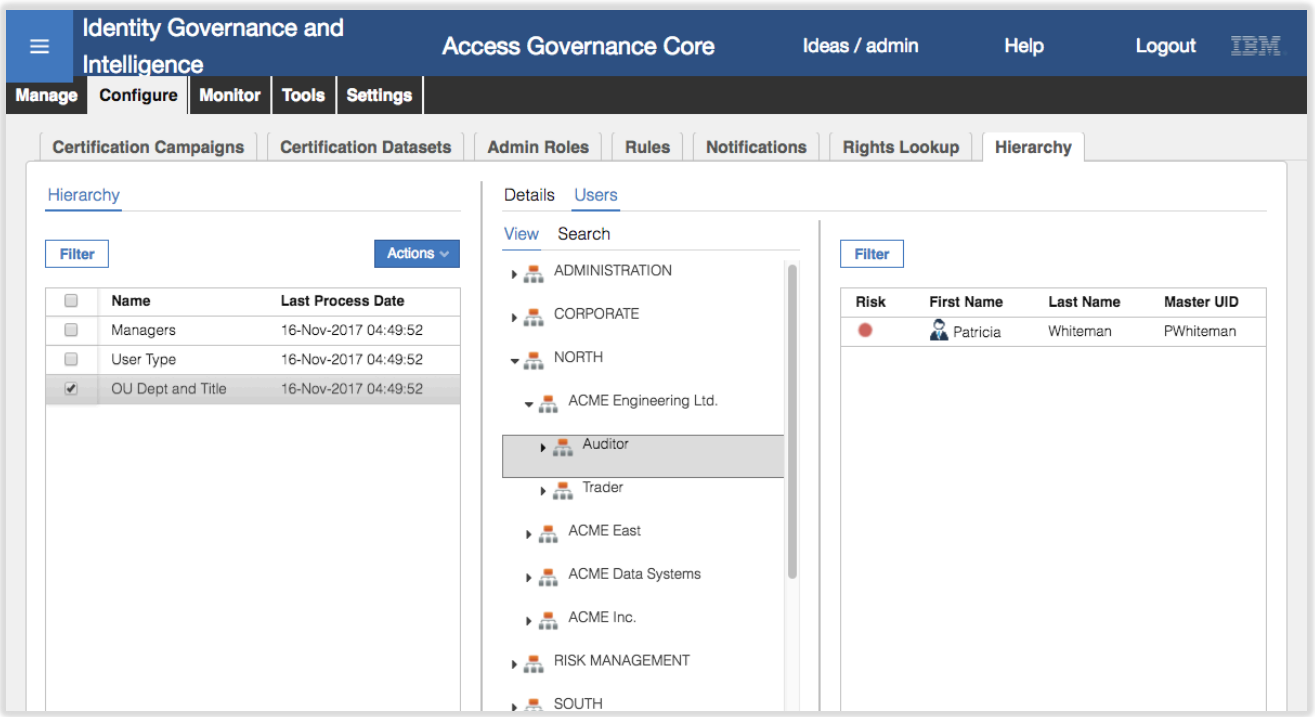
```
if (geo!=null && department!=null && title!=null ) {
   resultBean.setResultString(geo+";"+department+";"+title);
} else {
   resultBean.setResultString("root");
}
```

If each of geo, department and title have been found for the user, it sets the resultBean to the string “<geo>;<department>;<title>. Otherwise it sets it to “root”.

For example, if we look at Patricia Whiteman and the external data for her, we see a OU of NORTH, Department of “ACME Engineering Ltd.” and a Title of Auditor.



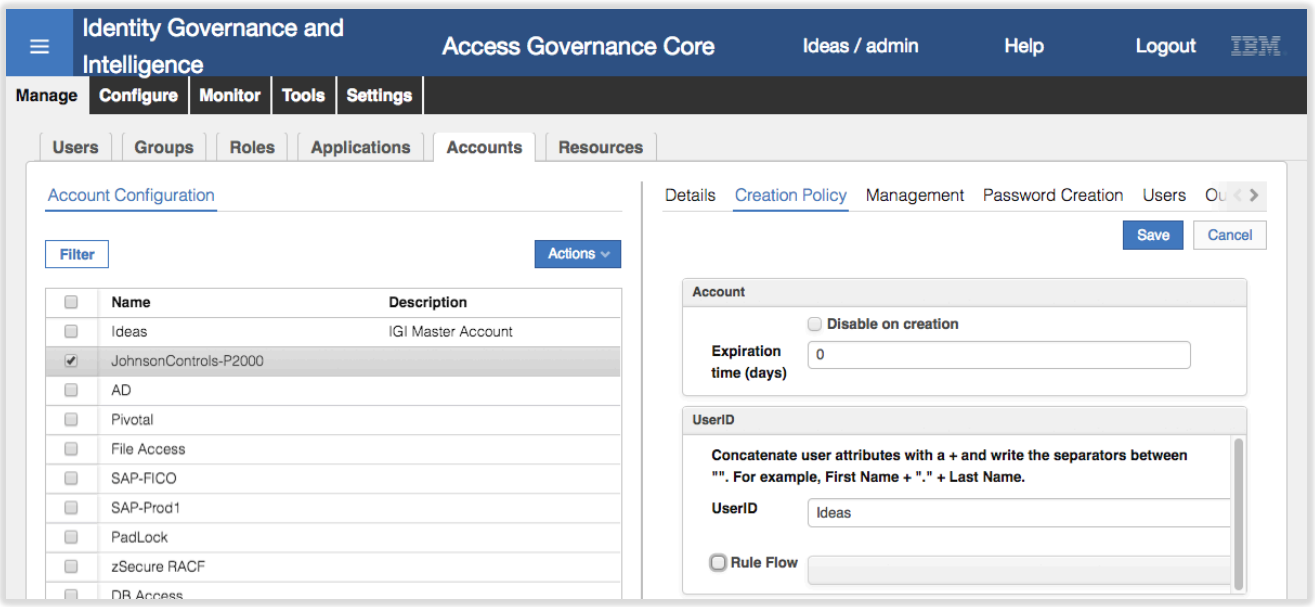
This rule with the appropriate hierarchy configuration results in the following output.



This mechanism could be used to build a hierarchy with any combination of user attributes.

5.3.3 Account Rules

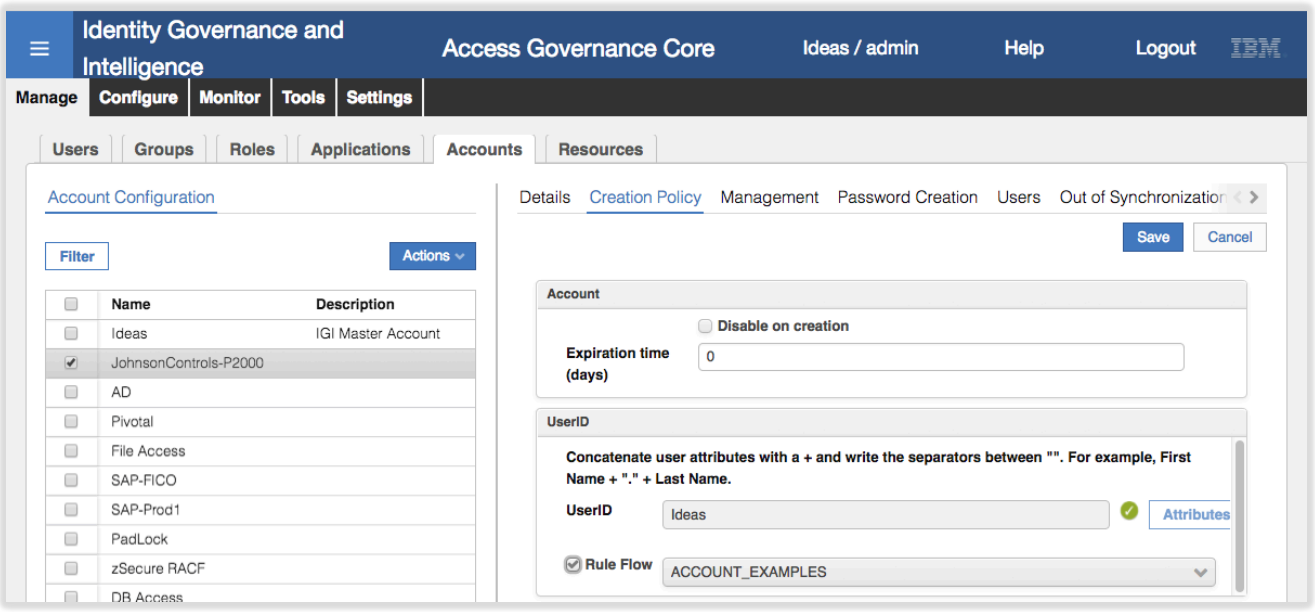
Account rules allow programmatic definition od a userid. For many deployments, the userid can be build using simple string concatenation in the UserID field of the Account configuration.



For example, putting `Name + "." Surname + "@" + MailProvider + "." + "com"` into the field will cause a userid like Patricia.Whiteman@ACME.com.

If more advanced construction logic is required, then rules of class Account can be used.

As with the other non-event rules, there must be a Rule Sequence of type Account defined, and this Rule Sequence must have one or more rules assigned to the Rule Flow (after creating them in the Rules Package). This rule flow is then assigned to the account configuration Creation Policy.



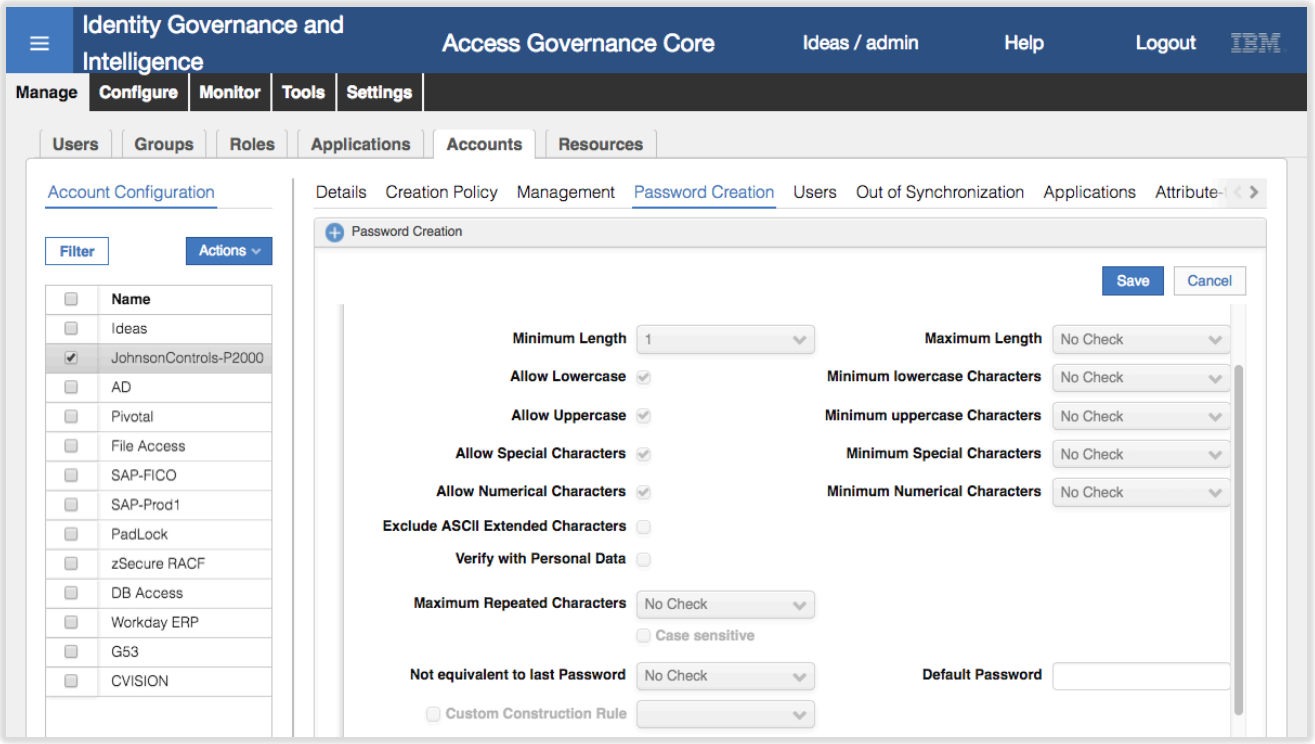
The working memory for account rules includes – TBC.

5.3.4 Example: Account Rules - TBA

To be added.

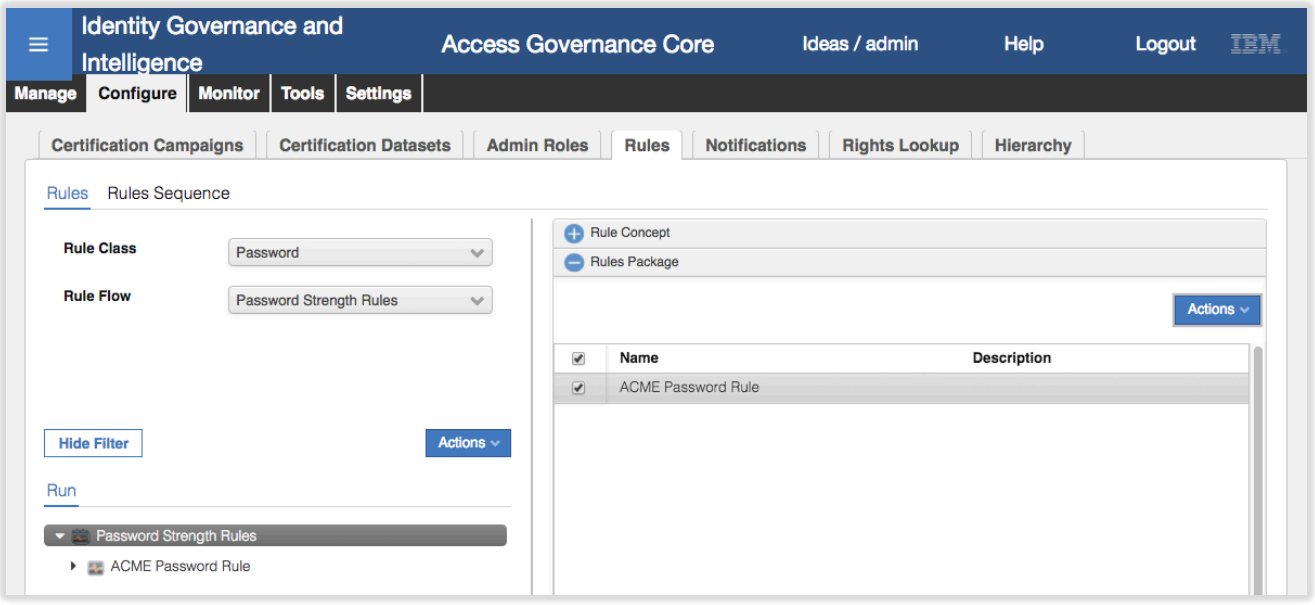
5.3.5 Password Rules

IGI provides a comprehensive set of password strength rules that include upper and lowercase characters, numbers and special characters. With IGI 5.2.3.1 there is a repeated character check.

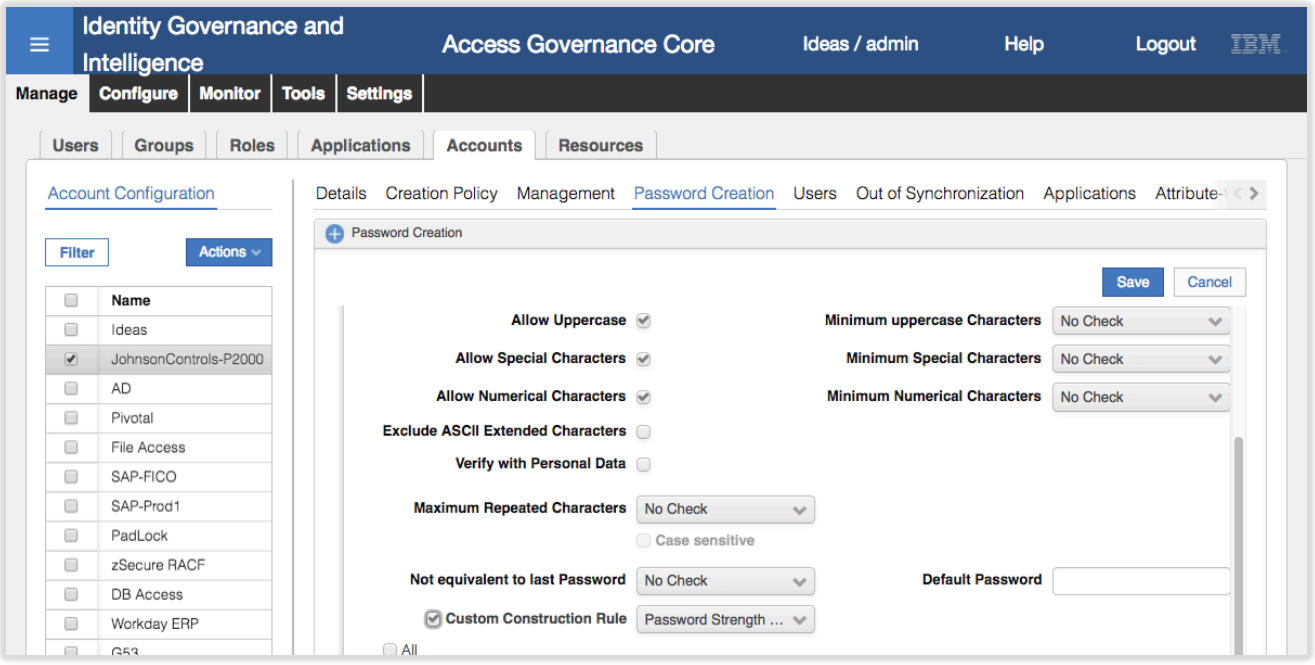


There is also the ability to cater for other requiements through the Custom Construction Rule option. This is a rule of class Password.

As with the other non-event rules, there must be a Rule Sequence of type Password defined, and this Rule Sequence must have one or more rules assigned to the Rule Flow (after creating them in the Rules Package).



This rule flow is then assigned to the account configuration Password Creation view.



The working memory for password rules includes – TBC.

5.3.6 Example: Password Rule - TBA

To be added.

5.4 Rules in Tasks and Jobs

Tasks and Jobs define scheduled background activity in IGI; a job is a unit of work, a task contains one or more jobs, and tasks run on one of IGIs schedulers.

Jobs can be:

- Provided out-of-the-box with IGI, many of them providing core product functionality,
- Developed as standalone Java programs using the EJB interface and loaded into IGI, or
- Rules run through a job provided for the purpose

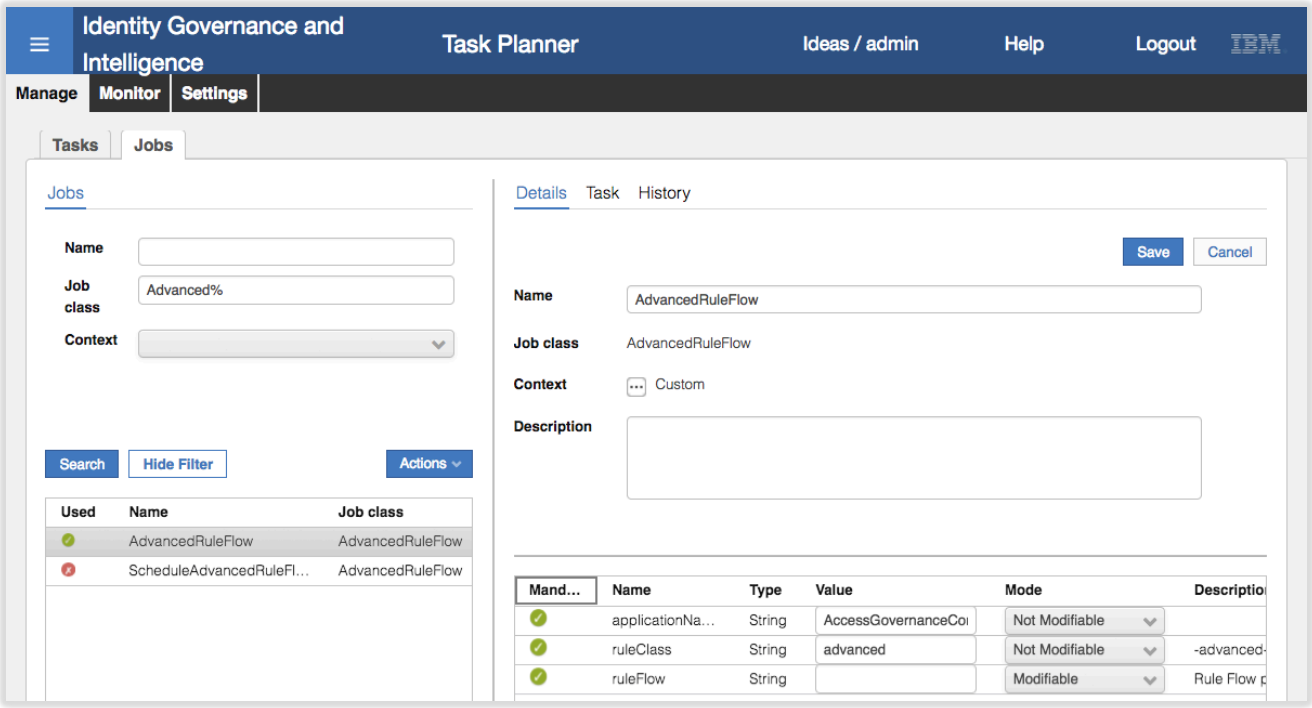
This section will look at the rules that can be scheduled.

5.4.1 Advanced Rules

Tasks, jobs and schedules are managed in the Task Planner module.

Jobs, Tasks and Scheduler for Advanced Rules

There is a job class of AdvancedRuleFlow for running Advanced rules as jobs.

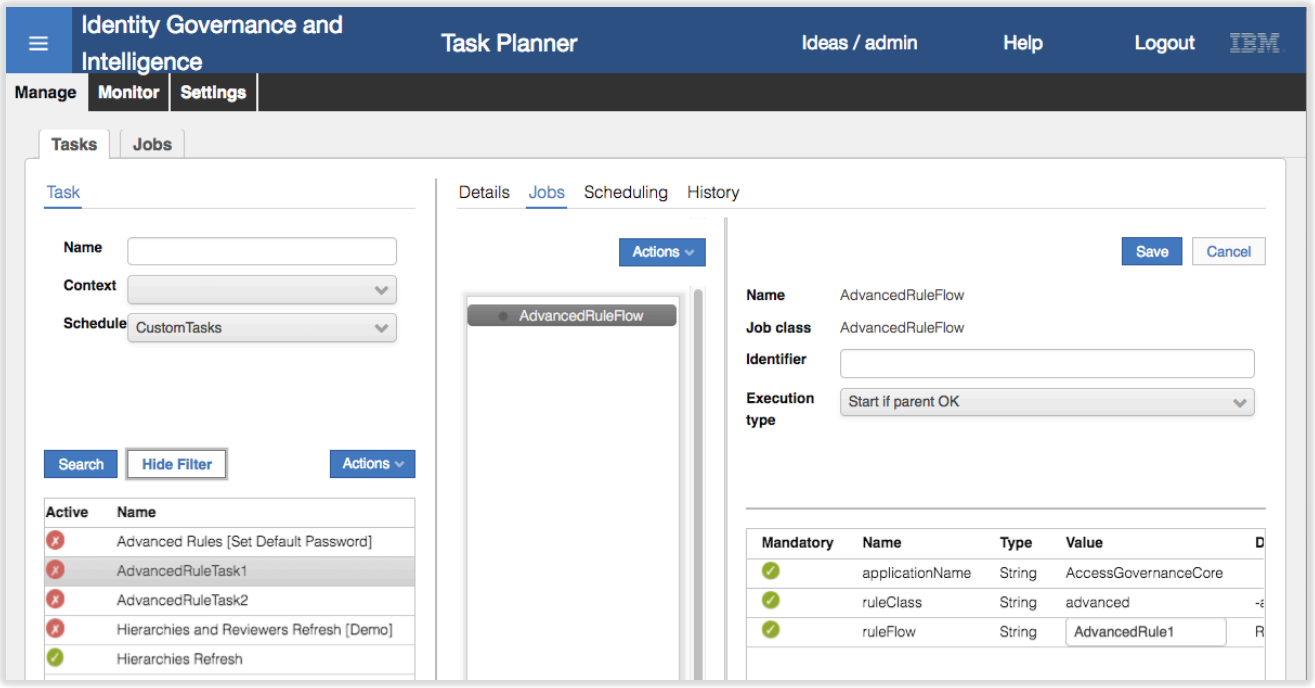


There are three settings for an AdvancedRuleFlow job:

- **applicationName** – the IGI application it is running under (this will probably be "AccessGovernanceCore")
- **ruleClass** – this can be either "advanced" for Advanced rules or "auth" for Authorization Digest" rules (this implies you can schedule rules to process revokes from a certification campaign)
- **ruleFlow** – this is the actual rule flow you want to run

The mode indicates whether the setting value can be specified when the job is associated with a task. This means you can have multiple tasks, all using the same job, but with different setting values.

For example, using the job shown above, I could use it in two different tasks (say AdvancedRuleTask1 and AdvancedRuleTask2) each using the same AdvancedRuleFlow job, but specifing different Advanced Rules to run.

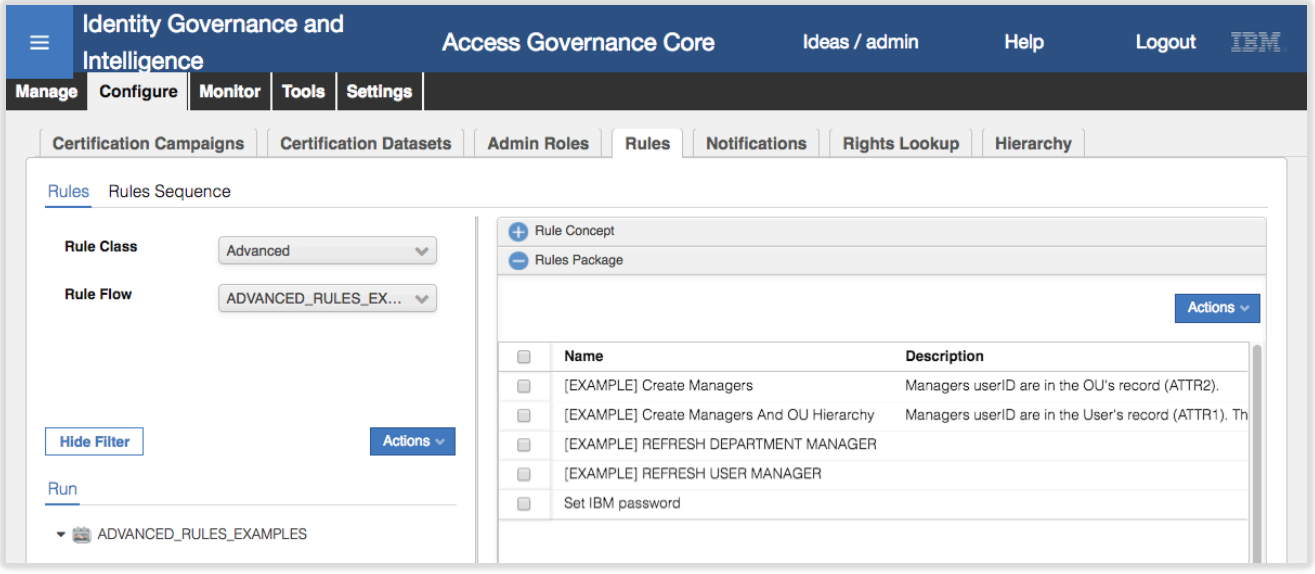


These tasks could run on any of the schedulers, but the standard approach is to use the Custom Tasks scheduler.

Advanced Rules

As with the other non-event rules, there must be a Rule Sequence of type Advanced defined, and this Rule Sequence must have one or more rules assigned to the Rule Flow (after creating them in the Rules Package).

These are managed in the Access Governance Core rules interface (**AGC > Configure > Rules**).



The rules could access any object in IGI, as the Package Import shows:

```
import java.sql.ResultSet
import java.sql.Statement

import com.engiweb.pm.dao.db.DAO
import com.engiweb.pm.entity.BeanList

import com.engiweb.profilemanager.common.bean.AccountBean
import com.engiweb.profilemanager.common.bean.ApplicationBean
import com.engiweb.profilemanager.common.bean.ExternalInfo
import com.engiweb.profilemanager.common.bean.OrgUnitBean
import com.engiweb.profilemanager.common.bean.UserBean
import com.engiweb.profilemanager.common.bean.UserErcBean
import com.engiweb.profilemanager.common.bean.entitlement.EntitlementBean
import com.engiweb.profilemanager.common.bean.event.EventBean
import com.engiweb.profilemanager.common.bean.event.EventInBean
import com.engiweb.profilemanager.common.ruleengine.action.OrgUnitAction
import com.engiweb.profilemanager.common.ruleengine.action.UserAction
import com.engiweb.profilemanager.common.ruleengine.action.UtilAction
import com.engiweb.profilemanager.common.ruleengine.action.reorganize._AccountAction
import com.engiweb.pm.entity.Paging
```

There are some examples shipped with IGI.

5.4.2 Example: Refresh Department Manager Admin Role

In this example, there is a Department Manager admin role. The role is assigned to any user who is flagged as a manager, and their scope is the OU that they are placed in.

This is achieving the same thing, defining members of the Department Manager admin role, as was done in an earlier rule example (Example: Set New User as Department Manager if A Manager on page 37).However that rule was driven by a single user change. This rule with completely rebuild the Department Manager admin role membership.

```
when
            eval ( true )
then
```

Unlike other rules we have looked at that fire when there are specified beans found, this rule will always run (`eval (true)`).

```
  String MANAGER_ROLE_NAME = "Department Manager";
  String IS_MANAGER_ATTR = "ATTR2"; // allowed values Y/N

  String SQL_QUERY_GET_MANAGERS =
    "select PM_CODE, OU from " +
          sql.getCntSQL().dbUser + ".USER_ERC " +
    "where deleted=0 and " + IS_MANAGER_ATTR + "='Y'";

  logger.debug("createManagers2 RULE START");
  UtilAction.setCodOperation(sql, "MR_advanced_createManagers2_" +
System.currentTimeMillis());
```

The first bit of code is setting some local variables:

- The MANAGER_ROLE_NAME is hardcoded as “Department Manager” (this needs to be the same as the actual admin role name).

- The IS_MANAGER_ATTR is the user attribute holding the Y/N flag to indicate whether the user is a manager or not (in the demo system it is ATTR2).
- The SQL_QUERY_GET_MANAGERS string is the query that will be run to find all users with the is_manager flag (ATTR2) set to "Y".

The query string includes both static text, the IS_MANAGER_ATTR variable and a method; `sql.getCntSQL().dbUser`. The sql.getCntSQL() object is used to get the container DB settings, such as the dbuser, which is also the default schema (thus resolving to IGACORE.USER_ERC).

It writes a debug message out and sets the operation code (MR_advanced_createManager2_<unique number>) to uniquely identify each execution.

```
// Get manager role object
EntitlementBean role = UtilAction.findEntitlementByName(sql, MANAGER_ROLE_NAME);
if (role == null) {
    throw new Exception("Role with name " + MANAGER_ROLE_NAME + " not found!");
}

// remove the role to all users
UtilAction.removeEntitlementToAllUsers(sql, role);
```

The next bit of code performs a lookup of the "Department Manager" admin role (`UtilAction.findEntitlementByName();`).

The last line (`UtilAction.removeEntitlementToAllUsers();`) will remove all users from ("to"?) the admin role. This emptying of the role is in preparation for building its membership from scratch.

```
// than reassign everything ...

// look for the managers
Statement stmt = sql.getCntSQL().getConnection().createStatement();
ResultSet rs = stmt.executeQuery(SQL_QUERY_GET_MANAGERS);
logger.debug("EXECUTE QUERY " + SQL_QUERY_GET_MANAGERS);
```

It defines a new SQL Statement object, and then uses the query string built above to determine all users with a is_manager attribute set to "Y". The code will resolve to something like:

"select PM_CODE, OU from IGCORE.USER_ERC where deleted=0 and ATTR2='Y'"

The results of the query, a set of userids (PM_CODE) and OUs for every user with the ATTR2=Y, are placed in a ResultSet object called rs.

```
// For each couple manager assign role and visibility
String managerCode = null;
String managedOUCode = null;
UserBean managerBean = null;
```

Before looping through the results, the code will create empty managerCode and managedOUCode strings, and an empty UserBean (managerBean) to represent the userid of the user, the OU the user is in, and the user object.

```
  while (rs.next()) {

      logger.debug("LOOP ...");

      managerCode = rs.getString(1);
      managedOUCode = rs.getString(2);

      // Get the userBean
      managerBean = UtilAction.findUserByCode(sql, managerCode);
      if (managerBean == null) {
          logger.error("Error, manager not found into IDEAS " + managerCode);
          continue;
      }
```

This is the start of the loop that will process all entries in the result set.

The code will save the two arguments from the result set, the userid and OU of the user.

It next searches for the user object for the user matching the userid and stores it in the managerBean object. If it's not found (`managerBean == null`) then an error is written out and the loop iterates to the next result (`continue;`).

```
      // assign the UM role
      OrgUnitBean currentOU = new OrgUnitBean();
      currentOU.setId(managerBean.getOrganizationalunit_id());

      BeanList roles = new BeanList();
      roles.add(role);

      // add role to OU
      OrgUnitAction.addRoles(sql, currentOU, roles, false);

      // add role to user
      UserAction.addRole(sql, managerBean, currentOU, roles, null, null, false, false);

      OrgUnitBean managedOU = UtilAction.findOrgUnitByCode(sql, managedOUCode);
      if (managedOU == null) {
          logger.error("Error OU not found into IDEAS " + managedOUCode);
          continue;
      }
```

The next bit of code will

- Create a new OU bean and set the Id based on the OU of the user.
- Create a new BeanList and adds the admin role ("Department Manager") entitlement bean to it
- Add the admin role to the OU (visibility) with the `OrgUnitAction.addRoles()` action
- Adds the admin role to the user with the `UserAction.addRole()` action
- Checks to see if the OU from the resultset (USER_ERC table for the user) and if it's not found the loop exits and continues with the next result. I don't see the point of this check as the code has already verified the OU from the user bean and you'd hope both have come from the DB table.

```
// Assign the visibility to the manager
      UtilAction.addResourcesToEmployment(sql, managerBean, role, managedOU);

  }
  rs.close();

  logger.debug("createManagers RULE END");
```

The last bit of code will set the scope of the user assignment to the "Department Manager" admin role (UtilAction.addResourcesToEmployment). It is setting the scope of their admin role to the OU where they reside.

Finally the results set object is closed.

This is a good example of a scheduled task that performs both bean-based and SQL query-based calls into IGI.

5.4.3 Example: Reset Ideas Account Passwords for All Users

This is a trivial example of a rule that may be useful in a PoC or demo scenario where you want to be able to easily reset all IGI account passwords back to a known value.

```
when
    eval( true )
then
  // Default Password
  String pwd = "Passw0rd!";

  // Ideas Account
  AccountBean ideasAccountBean = new AccountBean();
  ideasAccountBean.setPwdcfg_id(1L);
```

This rule is not reliant on a specific bean – it will run as part of a schedule (or on-demand), thus the condition of eval(true).

It sets a static password value (Passw0rd!), creates a new AccountBean and sets it to the id of the IGI account (Ideas account is always 1L).

```
  // Max 10000 Account
  Paging paging = new Paging(10000);

  BeanList<AccountBean> res = _AccountAction.findAccount(sql, ideasAccountBean, paging);
  for (AccountBean accountBean : res) {
    _AccountAction.changePwd(sql, "", pwd, accountBean);
  }
```

It sets a max search limit to 10,000 (`Paging(10000)`). It will then search for every Ideas account (based on the id set in the ideasAccountBean) using the `_AccountAction.findAccount();` method, but only up to 10,000 accounts.

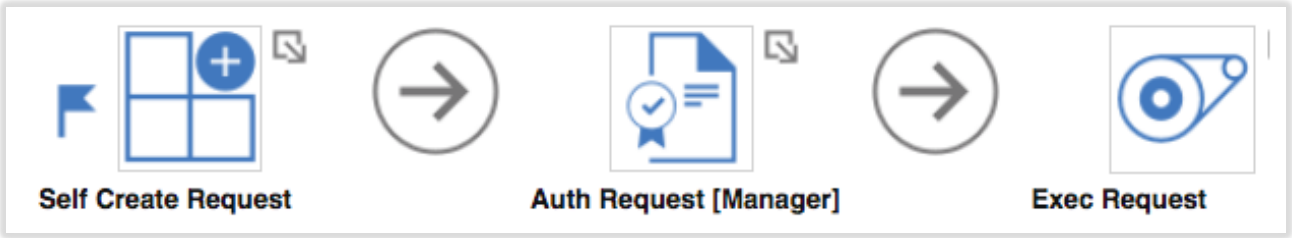
For each account found, it will change the password (`_AccountAction.changePwd();`) to the static password (pwd = "Passw0rd!").

5.5 Rules in Workflow Processes and Activities

Workflows drive access requests and associated activities, such as password management, account creation and role creation. IGI uses actvities (steps in a workflow) and processes (the workflow itself).

5.5.1 Use of Rules in Workflows

A typical access request workflow will have mutiple steps (activities); one to initiate the process (e.g. user requests an entitlement), one or more review steps (called Authorization) for different reviewers (e.g. user manager, then department manager) and an optional manual execute step (for manual provisioning where there is not an adapter or connector for that system).



Activities and processes can have configuration settings, but can also be extended through rules. We will not look at the activity and process configuration options, but will focus on the rules that can be used and how they can be used.

Processes and activities are managed in the Process Designer module. The rules for workflow can also be managed in the Process Designer (unlike most other rules that are managed in AGC).

Rules can be added to most activities as pre-action or post-action rules. Each activity has a small right-click icon to the top-right providing the ability to add or remove pre- and post-actions.

The term pre-action and post-action can be confusing, particularly as most activities involve building and presenting a form to a user in the IGI Service Center. The term pre-action does not mean “run this before presenting the form to the user”, it means “run this after the user has submitted the form but before the data is written to the database”. Similarly, the term post-action means “run this after the data has been written to the database”.

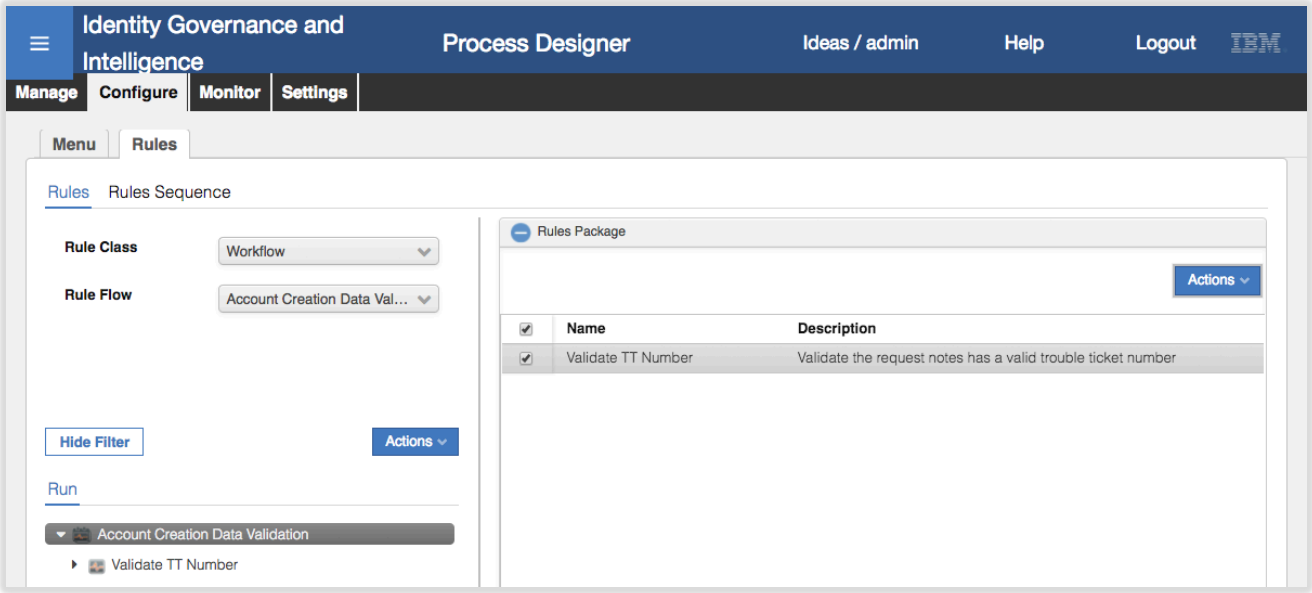
Choice of using pre-action vs. post-action can drive subsequent behaviour in IGI. For example, if you cancelled a change in a pre-action, there would be no historical record of it. Also, if you want to respond to the user in the Service Center UI, you need to do it in a pre-action (where you are still tied to the user session) rather than a post-action (where the user session is gone – the event is off the users view).

There are many uses for rules in workflow processes:

- Introduce some automatic rejection of a request based on some rules (e.g. some permissions flagged as not requestable, so if they get requested, automatically reject)
- Introduce non-linear flows in workflows. IGI workflows are a linear sequence of steps – there is no ootb branching or looping. However you can use rules to implement these mechanisms.
- Perform input data validation. The Service Center UI (via the process activities and some AGC settings) allows for limited data validation (e.g. data type, lookup list). If you need advanced data validation, you can do it in rules.
- In earlier versions of IGI, there was a need to develop rules for email notifications. This feature is now configurable via the UI, however there may still be a need for special emails to be sent.

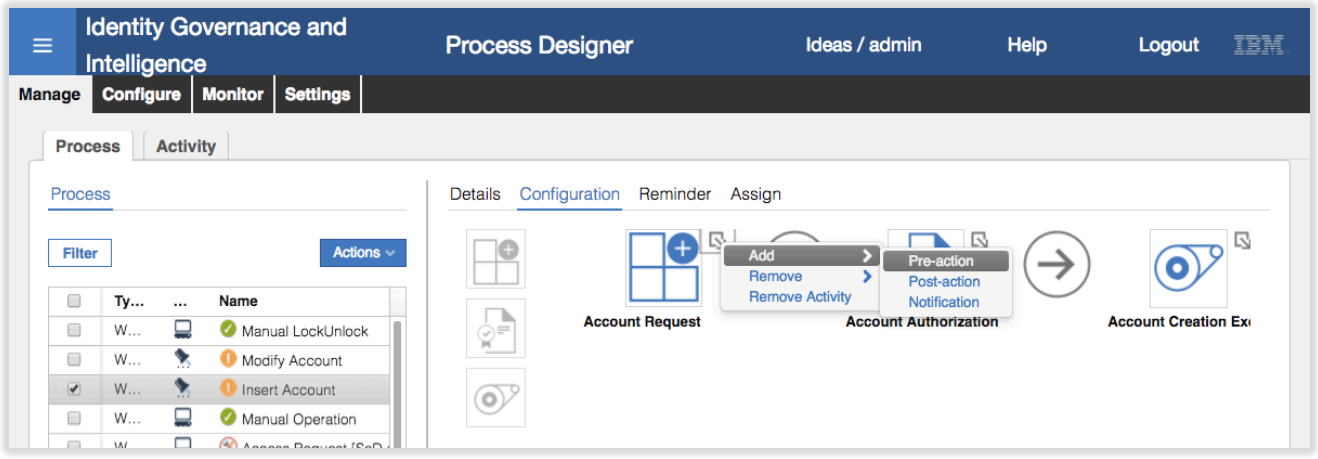
5.5.2 Adding Rules to Workflows

As with the other non-event rules, there must be a Rule Sequence of type Workflow defined, and this Rule Sequence must have one or more rules assigned to the Rule Flow (after creating them in the Rules Package). This can only be done in the Process Designer (**Process Designer > Configure > Rules**) and only the Workflow class can be selected.

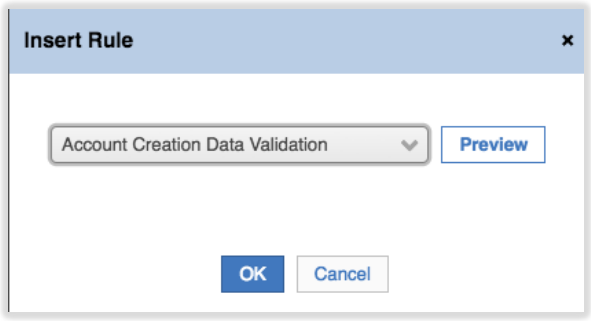


The rule flows are added to workflows by:

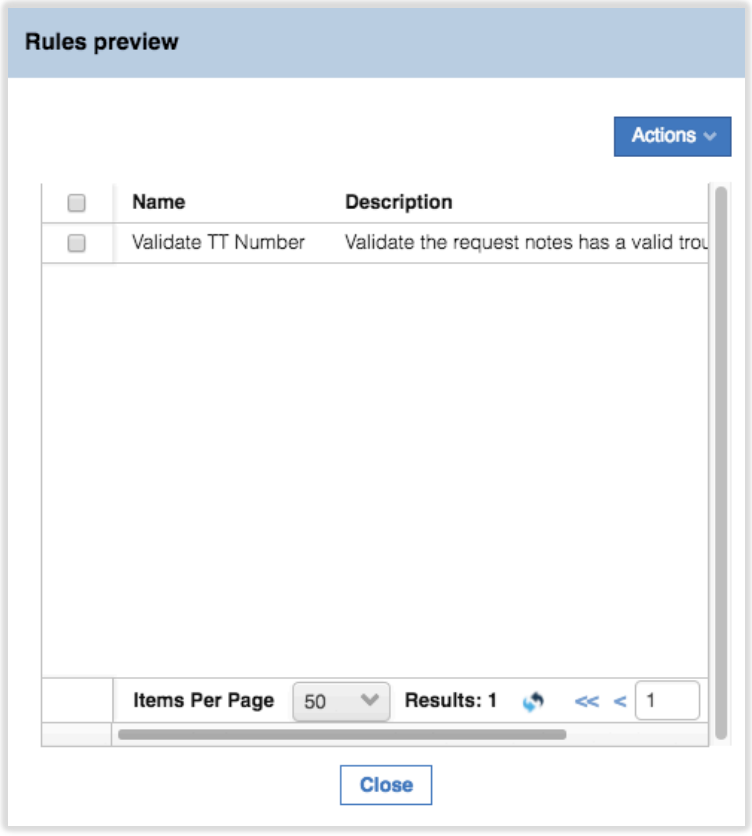
- Going to the **Process Designer > Manage > Process** and selecting the process
- Going to the Configuration tab
- Selecting, and right-clicking, the icon to the top-right of the relevant activity
- Selecting **Add > Pre-action** or **Add > Post-action**



- An **Insert Rule** dialog is displayed, where you select from one of the Workflow rule sequences

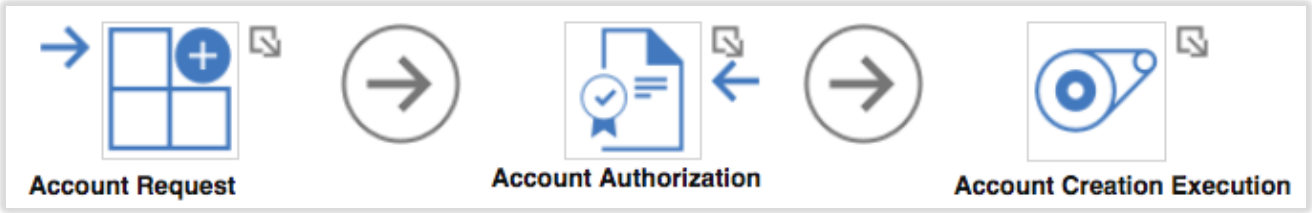


- You can use the **Preview** button to see the rules in the sequence



- You can select individual rules and use **Actions > View** to see the rule code
- Close the **Rules preview** dialg
- Click **OK** on the **Insert Rule** dialog to add this rule to the activity

A pre-action will show as a right arrow to the left of the activity icon. A post-action will show as a left arrow to the right of the activity icon. The following figure shows a pre-action for the Account Request activity and a post-action for the Account Authorization activity.



Rules added to workflow activities will be driven by the SwimRequestBean, however a lot of the detail, such as applicant details, beneficiary details and permission information is not available in the pre-action phase, only after the event is written to the database and IGI does its internal processing. You will also have access to the SwimEntitlementBean and EventOutBean.

When testing these rules, any logging will go to the `logs\iga_core\accessrequests.log` file.

5.5.3 Example: Workflow Rule to Check for Access Override (Pre-Action)

This rule came from a customer requirement where there was a need to flag some entitlements as blocked and not allow users to request them. This approach is an alternative to setting up visibility settings for entitlements against org units (or other attribute hierarchy groups).

To flag entitlements as blocked, one of the entitlement attributes (in this case “Tolerance”) was set to the string “BLOCK”. The rule is assigned as a pre-action.

```
when
    request : SwimRequestBean(  )
then
// Revoke request on tolerance Risk Level

logger.info("Request n. " + request.getId());

SwimRequestDAO swr = new SwimRequestDAO(logger);
swr.setDAO(sql);
request = swr.findRequestDetail(request);
```

This rule will operate on the SwimRequestBean() that contains all of the request information.

It logs the id of the request. Then it creates a new SwimRequestDAO (data access object), sets it to type of “sql” and pulls all of the request detail into the request (SwimRequestBean) object.

```
Integer reqStatus = request.getReqstatus();
if (reqStatus != null && reqStatus.equals(RequestStatus.ESCALATION.getCode())) {
```

This if statement checks to see 1) if there is a request status, and 2) if this is an ESCALATION request (i.e. that a risk was found and we're following the risk escalation process). The remainder of the code will only run if the two conditions are met.

```
	List<SwimEntitlementBean> listToRem = request.getRolesToRemove();
	List<SwimEntitlementBean> listToAdd = request.getRolesToAdd();

	logger.info("listToRem " + listToRem.size());
	logger.info("listToAdd " + listToAdd.size());

	SwimConverter2PM converter = new SwimConverter2PM();
	BeanList<EntitlementBean> entToAdd = converter.listSwim2PM(listToAdd);
	BeanList<EntitlementBean> entToRem = converter.listSwim2PM(listToRem);

	logger.info("EntToRem " + entToRem.size());
	logger.info("EntToAdd " + entToAdd.size());
```

This code extracts, and logs, the list of entitlements being removed and the list of entitlements being added. It then converts these lists into entitlement bean lists using the `converter.listSwim2PM` method (Swim refers to the Access Request Manager, whereas PM (i.e. Profile Manager) refers to the Access Governance Code, or AGC), so we are converting from an ARM event object to an AGC object (in this case SwimEntitlementBean to EntitlementBean – no idea why we need to).

```
	Long beneficiaryId = request.getBeneficiary_id();
	UserBean userBean = new UserBean();
	userBean.setId(beneficiaryId);

	CheckRiskDAO riskDao = new CheckRiskDAO(logger);
	riskDao.setDAO(sql);

	RiskInfoFull riskInfoFull = riskDao.checkUserFull(userBean, entToAdd, entToRem,
null, false);
	List<RiskBean> lRiskBean = riskInfoFull.getAfter().getAllRisk();
```

This block of code retrieves the beneficiary (who the request is for) id from the request and build a new local UserBean and sets the Id to the beneficiary Id.

It then creates a new CheckRiskDAO (data access object) to query the risk lists (`riskDao.checkUserFull();`) based on the user (userBean), the list of entitlements being added (entToAdd) and the list of entitlements being removed (entToRem). This results in a single object with all the risks for the user (after the changes are applied).

The last line builds a list of each unique risk as a result of the changes (`riskInfoFull.getAfter().getAllRisk();`). So, the lRiskBean is an array (List) of each risk after the changes are applied to the user.

```
	if (lRiskBean == null || lRiskBean.isEmpty()) {
		logger.info("No risk, skip! ");
		return;
	}
```

If nothing is found in the list of risks, the execution of the code (rule) ends – there is nothing to check. Otherwise it processes each risk in the lRiskBean list.

```
        for (RiskBean riskBean : lRiskBean) {

                RiskDAO lrb = new RiskDAO(logger);
                lrb.setDAO(sql);

                riskBean = (RiskBean) lrb.checkAB(riskBean, true);

                String currentTolerance = riskBean.getTolerance();

                logger.info("Current risk: " + riskBean.getName());
                logger.info("Current tolerance: " + currentTolerance);

                if (currentTolerance != null &&  currentTolerance.equals("BLOCK")) {

                        logger.info("BLOCK FOUND!");

                        UserBean userApplicantBean = new UserBean();
                        userApplicantBean.setCode(request.getApplicant_userid());

                        AuthorizeIncompatibility authDao = new
AuthorizeIncompatibility(logger);
                        authDao.setDAO(sql);

                        authDao.initialize("IDEAS", request.getEscalationPermission(),
userApplicantBean);
                        authDao.authorize(request, false);

                        return;
                }
        }
}
```

It builds a RiskDAO (data access object) to got get the details of the risk. The checkAB(); call builds a new RiskBean based on the current user risk (current iteration of lRiskBean).

It retrieves the tolerance attribute from the risk. If there is a value for tolerance and it's equal to "BLOCK", this risk has been flagged as a risk to block. It looks like the code will reject the change if ANY changes trigger a blocked risk, not just the specific change.

If flagged as blocked, it creates a new UserBean and sets it to the current applicant (i.e. person requesting the change). It creates a new AuthorizeIncompatibility data access object object, initialises it (not sure what the fields are) and sets the autorization to false (i.e. reject the request).

In summary this code:

- Check to ensure it's an escalation event from an access change request
- Gets the list of changes (entitlements to add and entitlements to remove) and converts them to entitlement beans
- Detemines any risks for this user as a result of the changes to entitlements
- If there are any risks for this user it processes each in turn, checks to see if the risk has a flag set to block the reuqest (risk attribute of Tolerance set to the string "BLOCK") and if so rejects all the changes.

5.5.4 Example: Set End Date for Risk-Inducing Request (Pre-Action)

This rule will look at a request for an entitlement, and if it generates a risk, will force an end-date on the entitlement.

The code is:

```
when
    requestBeanEnv : SwimRequestBean(  )
then
       logger.info("!!! Begin CheckRequestIncomp...:  " );

       // Beneficiary
       String beneficiaryUserid = requestBeanEnv.getBeneficiary_userid();
       // we need the UserBean
       UserBean user = new UserBean();
       user.setCode(requestBeanEnv.getBeneficiary_userid());
       BeanList<UserBean> users = UserAction.find(sql, user);
       user = users.get(0);
        logger.info("!!! User Considered:  " + user.getCode());
```

Triggers on a SwimRequestBean and sets it as requestBeanEnv.

It pulls the user id from the request with the `requestBeanEnv.getBeneficiary_userid();` call. It then builds a new user bean and sets the id (code) to be that from the request.

Finally, it does a search of all users with a matching userid. As the user code is (should be) unique in IGI, there will only be one user bean returned to the BeanList and so we can use the first record (`users.get(0);`).

```
    // we need the roles list to check risk
       SwimConverter2PM converter2PM = new SwimConverter2PM();
       BeanList<EntitlementBean> ee = new BeanList<EntitlementBean>();
       for (SwimEntitlementBean tmpBean : requestBeanEnv.getRolesToAdd()) {
         ee.add(converter2PM.SwimEntBean2EntBean(tmpBean));
       }
```

As with the previous example, the code needs to get the entitlement changes from the request to determine the risk changes.

It defines two objects; a SwimConverter2PM object to convert from a Swim list of entitlements to an AGC list of entitlements, and a BeanList of type EntitlementBean to hold the converted entitlements. For each added entitlement (`requestBenEnv.getRolesToAdd();`) it converts the Swim entitlement to an AGC entitlement in the ee BeanList.

```
    logger.info("!!! Before Risk Check..." );

    // Check risk information
       CheckRiskDirect riskD = new CheckRiskDirect();
       RiskInfo rI = riskD.checkUser(user, ee, null, null, null, sql);
```

This block of code defines a new direct method to access risk information.

> Note that this is a newer bit of code – it is using the Direct method to access data (ICheckRiskDirect in the JavaDoc), rather than the Data Access Object in the previous example. This is the preferred approach going forward.

The last line, `riskD.checkUser();`, will determine all risks (RiskInfo) for the user with the list of added entitlements. Unlike the previous example, any removed entitlements are not being considered.

```
        if (rI.getRiskNumber() > 0) {
          List<RiskBean> lRiskBean = rI.getAllRisk();
          for (RiskBean riskBean : lRiskBean) {
               logger.info("!!! Risk Name --> " + riskBean.getName());
               switch (riskBean.getRiskLevel().intValue()) {
                      case RiskInfo.LOW:
                             logger.info("!!! RISK LEVEL --> LOW");
                             break;
                      case RiskInfo.MEDIUM:
                             logger.info("!!! RISK LEVEL --> MEDIUM ");
                             break;
                      case RiskInfo.HIGH:
                             logger.info("!!! RISK LEVEL --> HIGH ");
                             break;
                      default:
               }// switch
               String riskType = riskBean.getRiskType_name();
               logger.info("!!! Risk Type --> " + riskType);
```

The next block of code checks there are risks (rI,getRisknumber() > 0) and if so processes the list. It defines a new list object and extracts all user risks found into it.

For each user risk, it logs the name and level) and then gets the riskType. For a vanilla IGI deployment this would be “SA” (Sensitive Access) or “SoD” (Separation of Duties), however custom ones may be added.

```
               if (riskType!=null && riskType.contains("SoD")) {
                      for (SwimEntitlementBean tmpBean2 : requestBeanEnv.getRolesToAdd()) {
                             logger.info("!!! Processing SWIMENT --> " +
tmpBean2.getName());
                             // Set new End Date
                             Calendar currentTime = Calendar.getInstance();
                             logger.info("!!! Current Time:  " + new SimpleDateFormat("dd-
MM-yyyy HH:mm:ss").format(currentTime.getTime()));
                             currentTime.add(Calendar.DAY_OF_MONTH, 1);
                             Date newEntDate = currentTime.getTime();
                             tmpBean2.setEnddate(newEntDate);
                             String stringDate = new SimpleDateFormat("dd-MM-yyyy
HH:mm:ss").format(newEntDate);
                             logger.info("!!! New End Date:  " + stringDate);
                      }
               }// if
          }// for
        } else {
               logger.info("!!! NO RISKS FOUND!" );
               return;
        }
        logger.info("!!! End CheckRequestIncomp");
```

It then checks if this is a SoD risk. If so, it gets all the entitlements being added, and for each one; gets the current date/time, adds one month to the current date, creates a new Data object, and sets the current entitlement end date to the new date (today + one month).

As with the previous example, this rule is not determining a specific entitlement triggering a risk (and setting the end date), rather it will set all requested entitlements to that end date if any new risks are found. I guess the assumption is that most requests will be fairly simple – one or two accesses per request. It should be possible to determine which requested entitlements are mapped to the business activities and this triggereing the risk change, and only apply the end date to them, but it would be significantly more coding.

5.5.5 Example: Second Level Approval Only for VV Requests (Post-Action)

This exemaple shows how a workflow step can be skipped based on logic in a rule.

```
when
    eventOut : EventOutBean(  )
then
// [ V1.1 - 2015-11-16 ]

// Who Authorize the skip step
String operator = "DPeak";
// Alias for auth step
String authAlias = "RG";

Long id;
try {
  String cOperation = eventOut.getCodiceOperazione();
  id = Long.valueOf(cOperation.substring((IdeasApplications.ACCESSREQUESTS.getAcronym() +
"_").length()));
} catch (Exception e) {
  return;
}
```

The first bit of code is setting some hardcoded strings; the person to authorise (approve) the skipped step (“DPeak”) and the alias for the authorize step (“RG”).

The next block is getting the operation code from the eventOut (EventOutBean) and using that to determine the request Id. It is performing a substring search on the operation code to find the text after the “IdeasApplication.ACCESSREQUESTS acronym” + “_”. Any event initiated by Access Request Manager will have an operation code of ARM_<requestid>, so this check is looking to see if the event contains the string “ARM_” in the operation code and storing the request Id in the id field.

```
SwimRequestBean swimReqBean = new SwimRequestBean();
swimReqBean.setId(id);
swimReqBean = RequestFindRule.findRequestDetail(sql, swimReqBean);

RequestStatus requestStatus = RequestStatus.get(swimReqBean.getReqstatus());
switch (requestStatus) {
case AUTHORIZABLE:
       break;
default:
       logger.error("Skip Request " + swimReqBean.getId() + ", status is " +
requestStatus);
       return;
}
```

The next block will create a new bean for the request and then get the details from the request detail using the id extracted from the operation code.

> I have no idea why this rule isn’t accessing the SwimRequestBean in the when/then clause to get the request detail.

It checks the request status to see if it’s “AUHORIZABLE” (i.e. ready for review/approval) and if not, exits the rule.

```
List<SwimEntitlementBean> roles;
RequestType reqType = RequestType.get(swimReqBean.getReqtype());
switch (reqType) {
case ROLE_ASSIGN:
       roles = swimReqBean.getRolesToAdd();
```

```
        break;
case ROLE_REMOVAL:
        roles = swimReqBean.getRolesToRemove();
        break;
case ROLE_RENEWAL:
        roles = swimReqBean.getRolesToUpdate();
        break;
default:
        logger.info("Skip Request " + swimReqBean.getId() + ", type is " + reqType);
        return;
}
```

The next block is getting the request type and building a roles list (List of type SwimEntitlementBean) based on whether we are adding, removing or renewing entitlements.

> It's not obvious why you would want to push for second level approval for removed roles if the use case is for visibility violations – it only makes sense when adding/renewing an entitlement where there is a visibility violation.

It may be possible to process requests that are ARM-generated, but not ROLE_ASSIGN/REMOVAL or RENEWAL. If so, this switch statement will cause the rule to exit for any other type.

```
// Check if the roles list includes a role in VV
for (SwimEntitlementBean swimEntBean : roles) {
  Long vv = swimEntBean.getVisibilityViolation();
  if (vv != null && vv.intValue() == SwimConstants.VISIBILITY_VIOLATION_ON) {

    logger.error("Request " + swimReqBean.getId() + " contains Role " +
swimEntBean.getNameI18n() + " in VV, follow next approval");
    // Found some role in VV exit
    return;
  }
}
```

For each role in the list look for a visibility violation. The if statement is checking for VV on this entitlement AND the visibility violation checking is enabled. If so, there is a VV and we need the workflow process to proceed to the next step in the workflow (i.e. second level approval).

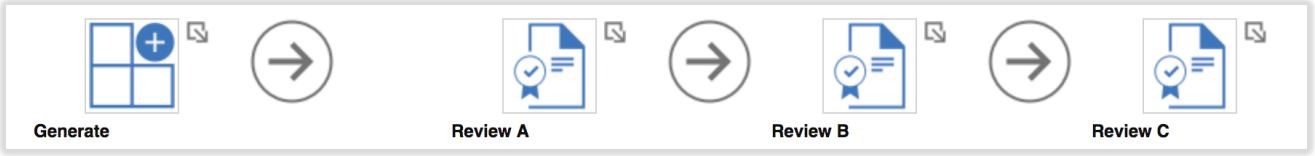
```
String nextStep = "AUTH/Auth ROwner$Access Request JBJ+Jump [Personal]";
logger.error("Set next Step: " + nextStep);

// No VV found, approve current reuqest
SwimRequestBean swimReqBean2 = RequestAuthorizationRule.authorizeRequest(sql, swimReqBean,
authAlias, nextStep, operator);

logger.error("Request " + swimReqBean2.getId() + "has been authorized, new staus is : " +
RequestStatus.get(swimReqBean2.getReqstatus()));
```

Otherwise (i.e. no VV found) we will force the second level approval activity to be skipped.

We do this by defining the next step (activity) in the workflow (process) – in this case its "AUTH/Auth ROwner$Access Request JBJ+Jump [Personal]".

Then the `RequestAuthorizationRule.authorizeRequest();` call is used to flag the next step as done and authorized. It will not show up on the second level manager list of actions as the rule has approved it and it would go to the next step in the workflow (or be provisioned to the target system).

This example shows how we can implement branching in workflow by using rules. Whilst IGI processes are a linear set of steps, you can define rules to skip activities based on logic by approving the next activity. For example, lets say you had three groups of reviewers (group A, group B and group C) and the reviewers that need to be involved depends on some runtime conditions, you could implement a process with five nodes: Generate –> Review A –> Review B –> Review C –> Execute.



Then you would write rules to determine the “skip” conditions and automatically approve the skipped activities. For example you might have a post-action on the Generate node that has logic to skip Review A and go straight to review B, and logic to skip both A and B to go to C.

This completes this example.

6 Scenario-Based Examples

Previous chapters have looked at how rules can be applied to different functions in IGI, and have provided examples of simple scenarios using rules. This chapter provides more complex examples that cover multiple functions or multiple rules to achieve a business use case.

6.1 Example: Certification Campaign Email Reminders and Expiration

The IGI certification campaign mechanism includes notifications. You can configure notifications for campaign start, campaign reaching some thresholds and campaign expiry.

However a customer needed a reminder mechanism as follows:

- If reviews are outstanding for five (5) days, send a reminder to the reviewer
- If reviews are outstanding for ten (10) days, send another reminder to the reviewer
- If reviews are outstanding for fifteen (15) days, send an escalation email
- If reviews are outstanding for eighteen (18) days, revoke the access

This could not be done with simple campaign notifications, it required a custom solution. The following sections will describe the solution.

6.1.1 Overview of Solution

The solution revolves around a scheduled task that runs daily to go search for any outstanding certification campaign items, and if they fall into the five/ten/fifteen/eighteen day periods, send an email as a reminder or revoke the access.

It involves:

1. Rules and a Rule Flow to extract the campaign items, check dates and perform the email/revoke,
2. A Task and Job to run the rule flow, and
3. Notification Templates to be used by the emails from the rules.

The following sections will describe each and how they are implemented, followed by some notes on testing and execution.

6.1.2 Rules and Rule Flow

There are four rules and a single rule flow of type “Advanced”.
The four rules are:

- CCReminder 05 days – to find outstanding items 5 days old and send email
- CCReminder 10 days – to find outstanding items 10 days old and send email
- CCReminder 15 days (Escalate Email) – to find outstanding items 15 days old and send email
- CCReminder 18 days (Revoke) – to find outstanding items 18 days old and revoke

These, and the imports, are explored in the following sections.

Rules Package Import

As with all rules in IGI, you need to include the Package Imports. For this example, these include a mix of the certification, profile manager (Access Governance Core), email, logging and utility classes.

They are are:

```
import com.crossideas.ap.common.ruleengine.action.RequestGenerationRule
import com.crossideas.certification.backend.dao.TemplateDAO
import com.crossideas.certification.common.AttestationRes
import com.crossideas.certification.common.bean.AttestationBean
import com.crossideas.certification.common.bean.TemplateBean
import com.crossideas.certification.common.enumeration.AttestationTypes
import com.engiweb.logger.impl.Log4JImpl
import com.engiweb.pm.dao.db.DAO
import com.engiweb.pm.entity.BeanList
import com.engiweb.pm.entity.Paging
import com.engiweb.pm.web.bean.AbstractBean
import com.engiweb.profilemanager.backend.dao.db.SQLH
import com.engiweb.profilemanager.common.bean.AccountBean
import com.engiweb.profilemanager.common.bean.ApplicationBean
import com.engiweb.profilemanager.common.bean.Block
import com.engiweb.profilemanager.common.bean.entitlement.EntitlementBean
import com.engiweb.profilemanager.common.bean.event.EventBean
import com.engiweb.profilemanager.common.bean.event.EventInBean
import com.engiweb.profilemanager.common.bean.ExternalInfo
import com.engiweb.profilemanager.common.bean.OrgUnitBean
import com.engiweb.profilemanager.common.bean.OrgUnitErcBean
import com.engiweb.profilemanager.common.bean.PwdCfgBean
import com.engiweb.profilemanager.common.bean.UserBean
import com.engiweb.profilemanager.common.bean.UserErcBean
import com.engiweb.profilemanager.common.enumerations.LockType
import com.engiweb.profilemanager.common.ruleengine.action.JobRoleAction
import com.engiweb.profilemanager.common.ruleengine.action.OrgUnitAction
import com.engiweb.profilemanager.common.ruleengine.action.reorganize._AccountAction
import com.engiweb.profilemanager.common.ruleengine.action.reorganize._OrgUnitAction
import com.engiweb.profilemanager.common.ruleengine.action.reorganize._UserAction
import com.engiweb.profilemanager.common.ruleengine.action.UserAction
import com.engiweb.profilemanager.common.ruleengine.action.UtilAction
import com.engiweb.toolkit.common.DBMSException
import java.sql.ResultSet
import java.sql.Statement
import java.util.HashMap
import java.util.Iterator
import java.sql.ResultSet
import java.util.ArrayList
import java.util.HashMap
import java.util.Map
import com.crossideas.email.common.action.WebEmailAction
import com.crossideas.email.common.bean.EmailDataBean
import com.engiweb.toolkit.common.enums.IdeasApplications
import com.crossideas.certification.backend.business.direct.PersonReviewDirect
import com.crossideas.certification.common.bean.EmploymentReviewBean

global com.engiweb.logger.impl.Log4JImpl logger
global com.engiweb.pm.dao.db.DAO sql
global com.engiweb.profilemanager.backend.dao.db.SQLH sql
```

There are probably a lot of unneccessary ones included, but it's a good set.

Rule – CCReminder 05 Days

This rule will find all outstanding items 5 days old and send an email.

```
when
    eval( true )
then
```

This will always run. This is the standard when/then clause for a scheduled rule as you want it to run always, and then use logic to determine the flow.

```
//
```

```
final String cc = "User Transfer Review";
final String EMAIL_TEMPLATE = "CCReminder";
final String MAIL_FROM = "CCReminder@acme.com";
```

This section of code sets three strings;

- The certification campaign that we are checking (in this case “User Transfer Review”),
- The email template to use (in this case “CCReminder”), and
- The account sending the email (in this case “CCReminder@acme.com)

```
StringBuilder sb = new StringBuilder();
sb.append("SELECT DISTINCT");
sb.append("          rev.NAME AS REVIEWER_NAME,");
sb.append("          rev.SURNAME AS REVIEWER_SURNAME,");
sb.append("          rev.CODE AS REVIEWER_CODE,");
sb.append("          rev.EMAIL AS REVIEWER_EMAIL,");
sb.append("          u.NAME AS USER_NAME,");
sb.append("          u.SURNAME AS USER_SURNAME,");
sb.append("          u.CODE AS USER_CODE ");
sb.append("FROM");
sb.append("          IGACORE.ATTESTATION a,");
sb.append("          IGACORE.EMPLOYMENT_REVIEW er,");
sb.append("          IGACORE.EMPLOYMENT_REVIEWER err,");
sb.append("          IGACORE.PERSON rev,");
sb.append("          IGACORE.PERSON u ");
sb.append("WHERE");
sb.append("          a.NAME = '"+ cc +"'");
sb.append("          AND a.ID = er.ATTESTATION");
sb.append("          AND er.ID = err.EMPLOYMENT_REVIEW");
sb.append("          AND err.CERT_FIRST_OWNER = rev.ID");
sb.append("          AND er.PERSON = u.ID");
sb.append("          AND er.REVIEW_STATE = 0");
sb.append("          AND (CURRENT_DATE BETWEEN (er.CREATION_DATE + 5) AND (er.CREATION_DATE
+ 6))");

String SQL_QUERY = sb.toString();

logger.debug("REMINDER 5 DAYS QUERY:\n" + SQL_QUERY);
```

This section of code is building the SQL Query (with standard SELECT…FROM…WHERE clauses) into a StringBuilder object.

It will pull rows from the ATTESTATION (campaign), EMPLOYMENT_REVIEW and EMPLOYMENT_REVIEWER (for campaign), and PERSON tables (once for user and once for reviewer), where the campaign name matches the string set above, review state = 0 (unprocessed) and the creation date is between 5 and 6 days ago.

It will return the user (being reviewed) and reviewer details. Notice that it is not collecting the entitlement being reviewed. This means that if there are multiple outstanding reviews for a single user there would be multiple rows returned, but as it is set as a “SELECT DISTINCT” there will only be a single row for each user with outstanding reviews.

Thus, this is focussed on just the users with outstanding review items, not the items themselves.

The StringBuilder object is converted to a string and is logged as a debug message.

```
ResultSet rs = sql.getCntSQL().getConnection().createStatement().executeQuery(SQL_QUERY);
```

The query is executed and results (i.e. users with outstanding campaign items between 5-6 days old) are stored in the results set. Each entry in the result set is processed in a loop.

```
while (rs.next()) {

	//String EMP_TO_REVIEW = rs.getString(1);
	String REVIEWER_NAME = rs.getString(1);
	String REVIEWER_SURNAME = rs.getString(2);
	String REVIEWER_CODE = rs.getString(3);
	String REVIEWER_EMAIL = rs.getString(4);
	//String ENTITLEMENT_TO_REVIEW = rs.getString(6);
	String USER_NAME = rs.getString(5);
	String USER_SURNAME = rs.getString(6);
	String USER_CODE = rs.getString(7);

	// Add users to send out notifications
	ArrayList<String> recipients = new ArrayList<String>();
	recipients.add(REVIEWER_EMAIL);

	Map map = new HashMap();
	map.put("$P{attestation.campaign.recipient.name}", REVIEWER_NAME);
	map.put("$P{attestation.campaign.recipient.surname}", REVIEWER_SURNAME);
	map.put("$P{email}", REVIEWER_EMAIL);
	map.put("$P{details}", "Involved User: " + USER_NAME + " " + USER_SURNAME + " (" +
USER_CODE + ")");

	EmailDataBean emailBean = new EmailDataBean(MAIL_FROM, recipients);

	WebEmailAction.submitEmail(sql, map, "", "admin", "EN", "EN",
IdeasApplications.EMAILSERVICE.getName(), EMAIL_TEMPLATE, emailBean);

	logger.info("Sent an Email to: " + REVIEWER_NAME + "-" + REVIEWER_SURNAME + "-" +
REVIEWER_CODE + "-" + REVIEWER_EMAIL);
	logger.info("5 days no action on: " +  USER_NAME + "-" + USER_SURNAME + "-" +
USER_CODE);
}
rs.close();
```

The result set rows consist of an array of string objects, one for each column in the SELECT clause (REVIEWER_NAME, REVIEWER_SURNAME, REVIEWER_CODE, REVIEWER_EMAIL, USER_NAME, USER_SURNAME, and USER_CODE). These are stored in individual strings for the email template.

> If you want to use this rule and SQL query, and you change the SELECT statement, you must change the corresponding rs.getString() statements.

The rest of this block of code is very similar to earlier email notification rules. It creates an array of recipients and puts the reviewer email in it. It sets some email template variables; `attestation.campaign.recipient.name`, `attestation.campaign.recipient.surname`, `email` (for reviewer) and `details` for the user being reviewed)). It creates a new EmailDataBean with the MAIL_FROM (string set at the top) and array of recipients (the reviewer).

Then it uses the WebEmailAction.submitEmail() action to send the email using the EMAIL_TEMPLATE variable set at the top (“CCReminder”).

The last two lines write info messges to the log with details of the recipient and user.

The loop repeats for every user found. This concludes this rule.

Rule – CCReminder 10 Days

This rule will find all outstanding items 10 days old and send an email.

The rule code is exactly the same as the CCReminder 05 Days rule with one exception; the selection clause for the date…

```
sb.append("SELECT DISTINCT");
sb.append("         rev.NAME AS REVIEWER_NAME,");
sb.append("         rev.SURNAME AS REVIEWER_SURNAME,");
sb.append("         rev.CODE AS REVIEWER_CODE,");
sb.append("         rev.EMAIL AS REVIEWER_EMAIL,");
sb.append("         u.NAME AS USER_NAME,");
sb.append("         u.SURNAME AS USER_SURNAME,");
sb.append("         u.CODE AS USER_CODE ");
sb.append("FROM");
sb.append("         IGACORE.ATTESTATION a,");
sb.append("         IGACORE.EMPLOYMENT_REVIEW er,");
sb.append("         IGACORE.EMPLOYMENT_REVIEWER err,");
sb.append("         IGACORE.PERSON rev,");
sb.append("         IGACORE.PERSON u ");
sb.append("WHERE");
sb.append("         a.NAME = '"+ cc +"'");
sb.append("         AND a.ID = er.ATTESTATION");
sb.append("         AND er.ID = err.EMPLOYMENT_REVIEW");
sb.append("         AND err.CERT_FIRST_OWNER = rev.ID");
sb.append("         AND er.PERSON = u.ID");
sb.append("         AND er.REVIEW_STATE = 0");
sb.append("         AND (CURRENT_DATE BETWEEN (er.CREATION_DATE + 10) AND
(er.CREATION_DATE + 11))");
```

All other code is the same. It builds the same email variables and uses the same email template (“CCReminder”) as the previous rule.

Rule – CCReminder 15 Days (Escalate Email)

This rule will find all outstanding items 15 days old and send an email to an escalation reviewer. This escalation reviewer is defined in the ATTR12 attribute on the user ERC record. This is an installation-specific setting for a specific customer deployment. The value may be the user manager, a supervisor, a department manager or secretary or similar. The important thing is that it’s storing the userid of this escalation reviewer on the person and using that to pull this reviewers details to send an email to.

The code is very similar to the two previous rules, with some specific differences.

```
when
    eval( true )
then
//
final String cc = "User Transfer Review";
final String EMAIL_TEMPLATE = "CCReminderEscalate";
final String MAIL_FROM = "CCReminder@acme.com";
```

The only difference in the variables being set is the EMAIL_TEMPLATE – it is using “CCReminderEscalate” rather than the “CCReminder” template in the earlier two rules.

```
StringBuilder sb = new StringBuilder();
sb.append("SELECT DISTINCT");
sb.append("         rev.NAME AS REVIEWER_NAME,");
```

```
sb.append("          rev.SURNAME AS REVIEWER_SURNAME,");
sb.append("          rev.CODE AS REVIEWER_CODE,");
sb.append("          rev.EMAIL AS REVIEWER_EMAIL,");
sb.append("          mrev.NAME AS MREVIEWER_NAME,");
sb.append("          mrev.SURNAME AS MREVIEWER_SURNAME,");
sb.append("          ue.ATTR12 AS MREVIEWER_CODE,");
sb.append("          ue.ATTR13 AS MREVIEWER_EMAIL,");
sb.append("          u.NAME AS USER_NAME,");
sb.append("          u.SURNAME AS USER_SURNAME,");
sb.append("          u.CODE AS USER_CODE ");
sb.append("FROM");
sb.append("          IGACORE.ATTESTATION a,");
sb.append("          IGACORE.EMPLOYMENT_REVIEW er,");
sb.append("          IGACORE.EMPLOYMENT_REVIEWER err,");
sb.append("          IGACORE.USER_ERC ue,");
sb.append("          IGACORE.PERSON mrev,");
sb.append("          IGACORE.PERSON rev,");
sb.append("          IGACORE.PERSON u ");
sb.append("WHERE");
sb.append("          a.NAME = '"+ cc +"'");
sb.append("          AND a.ID = er.ATTESTATION");
sb.append("          AND er.ID = err.EMPLOYMENT_REVIEW");
sb.append("          AND err.CERT_FIRST_OWNER = rev.ID");
sb.append("          AND er.PERSON = u.ID");
sb.append("          AND ue.ID = rev.USER_ERC");
sb.append("          AND ue.ATTR12 = mrev.CODE");
sb.append("          AND er.REVIEW_STATE = 0");
sb.append("          AND (CURRENT_DATE BETWEEN (er.CREATION_DATE + 15) AND
(er.CREATION_DATE + 16))");

String SQL_QUERY = sb.toString();

logger.debug("REMINDER TO Mana 10 DAYS QUERY:\n" + SQL_QUERY);
```

The SQL query is pulling in the escalation reviewer (“MREVIEWER***”) details as well as the details for the reviewer and user being reviewed. It also has a different date search argument.

```
ResultSet rs = sql.getCntSQL().getConnection().createStatement().executeQuery(SQL_QUERY);
while (rs.next()) {

       String REVIEWER_NAME = rs.getString(1);
       String REVIEWER_SURNAME = rs.getString(2);
       String REVIEWER_CODE = rs.getString(3);
       String REVIEWER_EMAIL = rs.getString(4);
       String MREVIEWER_NAME = rs.getString(5);
       String MREVIEWER_SURNAME = rs.getString(6);
       String MREVIEWER_CODE = rs.getString(7);
       String MREVIEWER_EMAIL = rs.getString(8);
       String USER_NAME = rs.getString(9);
       String USER_SURNAME = rs.getString(10);
       String USER_CODE = rs.getString(11);

       // Add users to send out notifications
       ArrayList<String> recipients = new ArrayList<String>();
       recipients.add(MREVIEWER_EMAIL);

       Map<String, String> map = new HashMap<String, String>();
       map.put("$P{attestation.campaign.recipient.name}", REVIEWER_NAME);
       map.put("$P{attestation.campaign.recipient.surname}", REVIEWER_SURNAME);
       map.put("$P{mname}", MREVIEWER_NAME);
       map.put("$P{msurname}", MREVIEWER_SURNAME);
       map.put("$P{details}", "15 Days passed, and Reviewer did not processed tickets.
Involved User: " + USER_NAME + " " + USER_SURNAME + " (" + USER_CODE + ")");

       EmailDataBean emailBean = new EmailDataBean(MAIL_FROM, recipients);
```

```
        WebEmailAction.submitEmail(sql, map, "", "admin", "EN", "EN",
IdeasApplications.EMAILSERVICE.getName(), EMAIL_TEMPLATE, emailBean);

        // logger lines removed for clarity
}
rs.close();
```

As in earlier rules it runs the query and then for every user-reviewer-escalationreviewer result it; pulls the column values from the result, creates the recipient list (based on the escalation reviewer, not the reviewer as earlier), sets the email template variable values, then uses the WebMailAction.submitEmail() action to send the email using the "CCReminderEscalate" template.

So, with this rule it will pull the user, reviewer and escalation reviewer from the DB where the creation data is between 15-16 days and the item hasn't been reviewed, and for each unique user-reviewer-escalationreviewer result it will send an email (using the "CCReminderEscalate" template) to the escalation reviewer.

CC Reminder 18 Days (Revoke)

This rule will find all outstanding items 18 days old and automatically revoke them. The code is similar to the previous rules.

```
when
    eval( true )
then
//
final String cc = "User Transfer Review";
```

Unlike the earlier rules, we don't need to set email template or sender in variables as there is no email being sent.

```
StringBuilder sb = new StringBuilder();
sb.append("SELECT DISTINCT");
sb.append("          er.ID AS EMP_TO_REVIEW,");
sb.append("          rev.CODE AS REVIEWER_CODE,");
sb.append("          u.CODE AS USER_CODE,");
sb.append("          ent.NAME AS ENT_NAME ");
sb.append("FROM");
sb.append("          IGACORE.ATTESTATION a,");
sb.append("          IGACORE.EMPLOYMENT_REVIEW er,");
sb.append("          IGACORE.EMPLOYMENT_REVIEWER err,");
sb.append("          IGACORE.PERSON rev,");
sb.append("          IGACORE.ENTITLEMENT ent,");
sb.append("          IGACORE.PERSON u ");
sb.append("WHERE");
sb.append("          a.NAME = '"+ cc +"'");
sb.append("          AND a.ID = er.ATTESTATION");
sb.append("          AND er.ID = err.EMPLOYMENT_REVIEW");
sb.append("          AND err.CERT_FIRST_OWNER = rev.ID");
sb.append("          AND er.PERSON = u.ID");
sb.append("          AND er.ENTITLEMENT = ent.ID");
sb.append("          AND er.REVIEW_STATE = 0");
sb.append("          AND (CURRENT_DATE BETWEEN (er.CREATION_DATE + 18) AND
(er.CREATION_DATE + 20))");

String SQL_QUERY = sb.toString();

logger.debug("Rvoke Entitlement More than 18 DAYS QUERY:\n" + SQL_QUERY);
```

The SQL query being built is simpler; it will retrieve the employment, user and entitlement, and the reviewer code, for items between 18-20 days old and not reviewed.

The query needs to include an additional table, IGACORE.ENTITLEMENT, and use this to get the entitlement name (ent.NAME AS ENT_NAME) when the entitlement id (er.ENTITLEMENT on the IGACORE.ENTITLEMENT_REVIEW) matches the id in the ENTITLEMENT table.

```
PersonReviewDirect pRD = new PersonReviewDirect();

ResultSet rs = sql.getCntSQL().getConnection().createStatement().executeQuery(SQL_QUERY);
while (rs.next()) {

	long EMP_TO_REVIEW = rs.getLong(1);
	String REVIEWER_CODE = rs.getString(2);
	String USER_CODE = rs.getString(3);
	String ENT_NAME = rs.getString(4);

	EmploymentReviewBean erb = new EmploymentReviewBean();
	erb.setId(EMP_TO_REVIEW);

	BeanList<EmploymentReviewBean> lerb = new BeanList<>();
	lerb.add(erb);

	logger.info("Revoking " + ENT_NAME + " to " + USER_CODE);

	pRD.revoke(lerb, "18 days passed: Revoked by System", REVIEWER_CODE, sql);
}
rs.close();
```

It creates a new PersonReviewDirect object for the entitlement to be revoked (which will be reused for each iteration).

As earlier it runs the query and the resulting rows (in the result set) are processed individually in a loop.

The loop will pull the employment (user-entitlement mapping), reviewer (code), user (code) and entitlement name from the query result. A new EmploymentReviewBean is created and set to the id of the employment being reviewed and this is added to a BeanList.

The pRD.revoke() method will revoke the access with a message of “18 days passed: Revoked by System” against the reviewer.

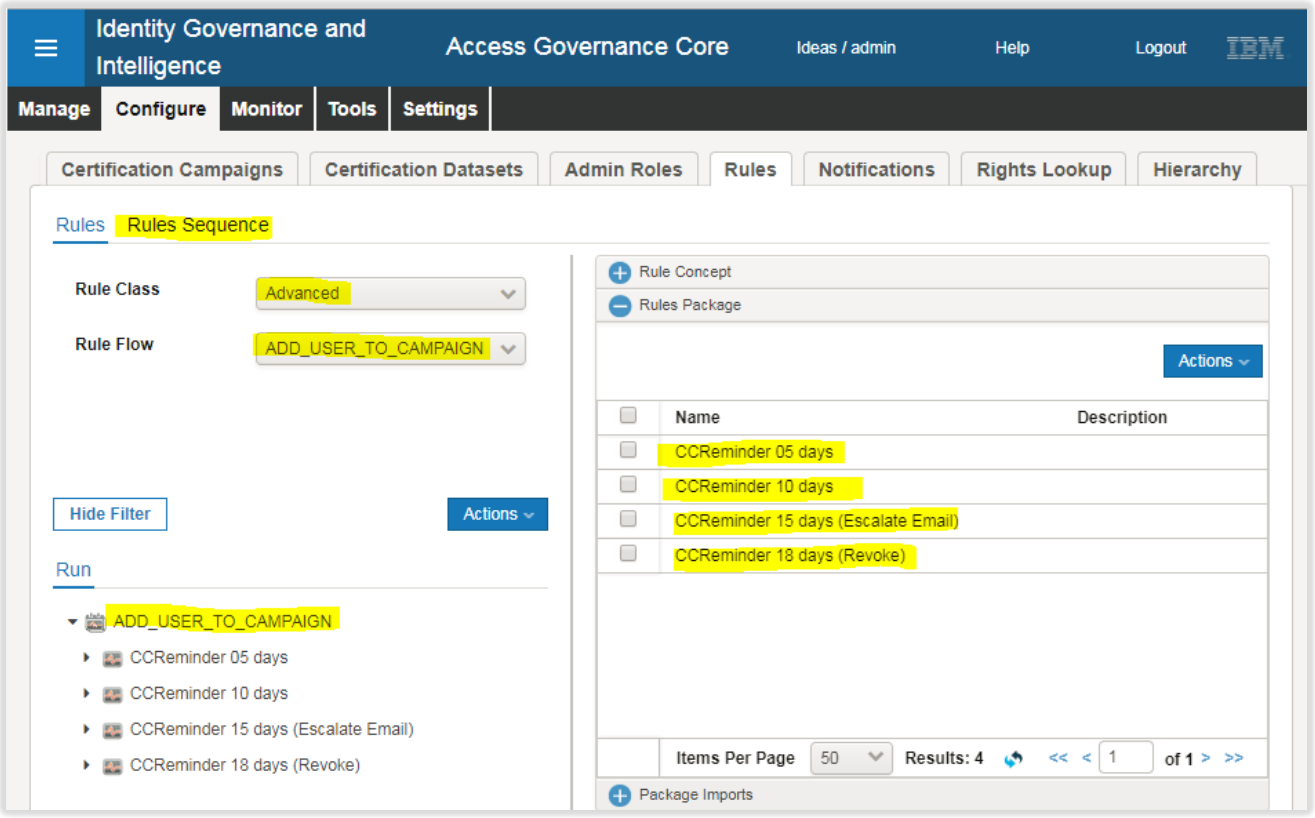
As with any rules, these should have the imports included and the rules should be verified (Actions > Verify) to check for any coding errors.

Rule Flow

A single rule flow of class “Advanced” is created in the Rules interface. Recall that the Advanced rule class is used for rules that will be attached to tasks and jobs.

In this case the rule flow is called “ADD_USER_TO_CAMPAIGN”.

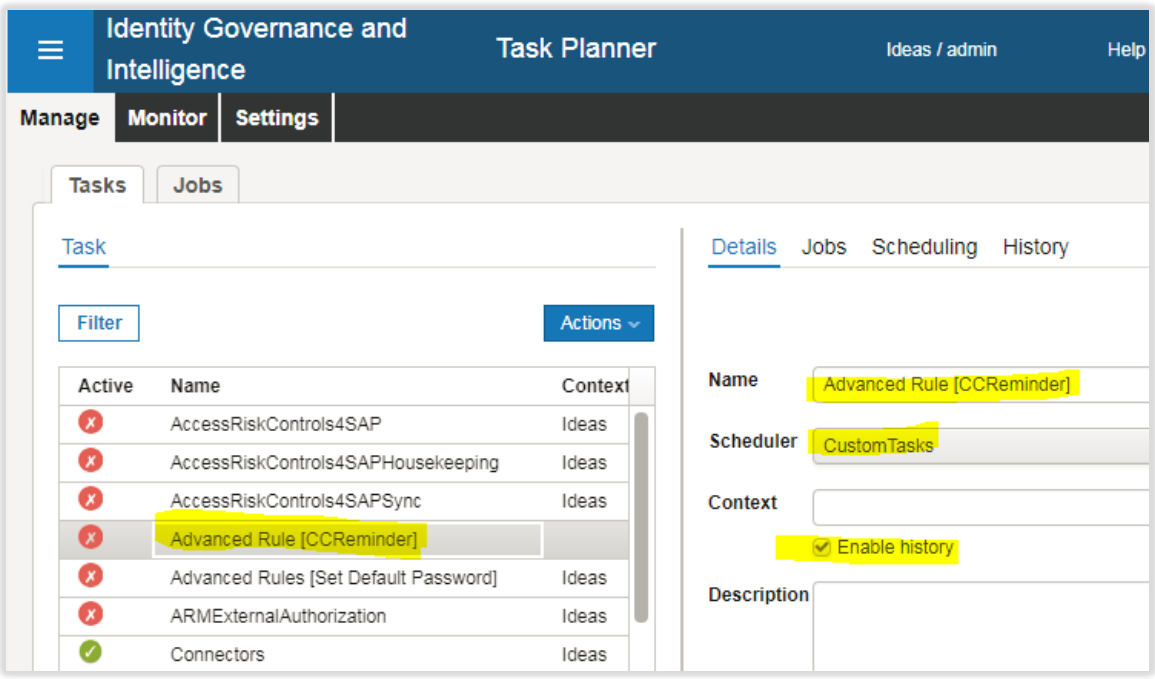
Recall that rules in a rule flow are processed sequentually, so an earlier rule may impact the execution of a later rule. In this case each rule is independent of the earlier one. So the rules could be added to the flow in any order. To keep it simple, they have been added to the flow in the order of the date checks.



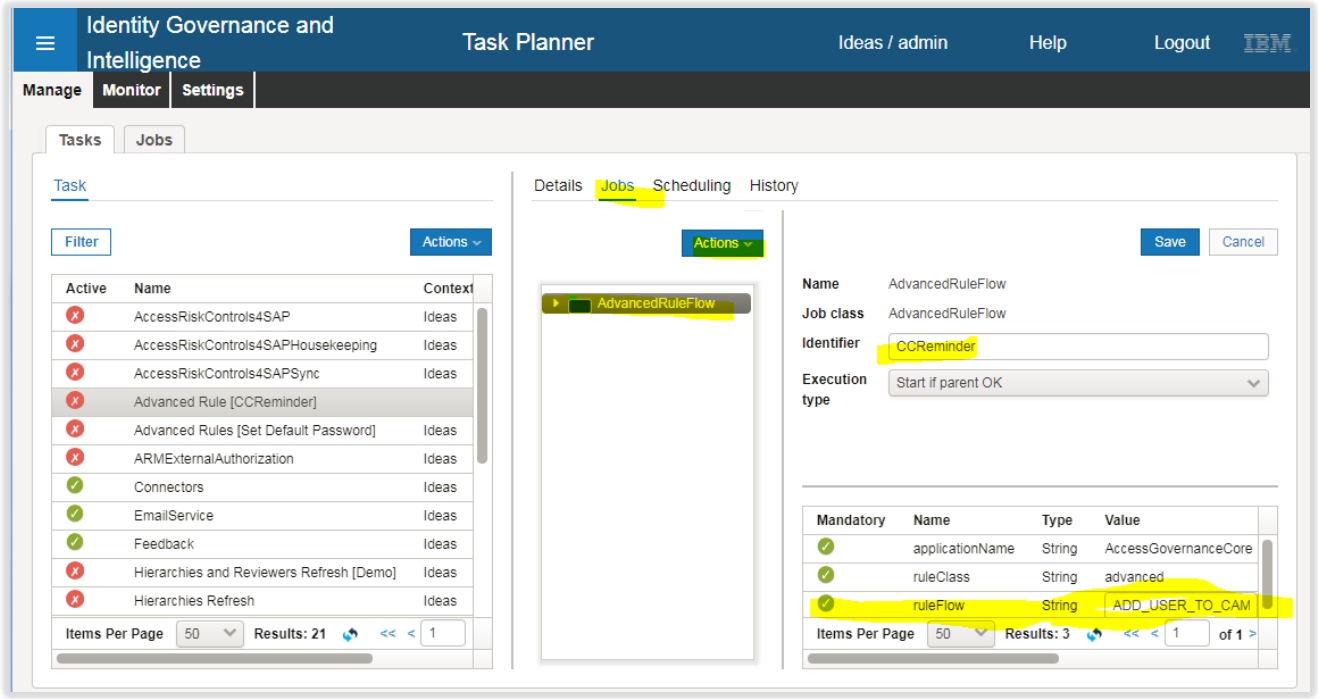
Now that the rules have been defined and an Advanced rule flow configured with the four rules, we can attach it to a task.

6.1.3 Tasks and Jobs for Custom Rule Flow

This new rule flow is executed via a schedule based on a Task with a Job. A new task is created in the Task Planner, similar to what is shown below.



For this new Task we have a single Job is created, similar to the following.



It is important that the ruleFlow attribute value is exactly the same as the name of the rule flow from above (e.g. “ADD_USER_TO_CAMPAIGN”). You can name the rules and rule flow whatever you want, but the job must be configured to use the correct rule flow.

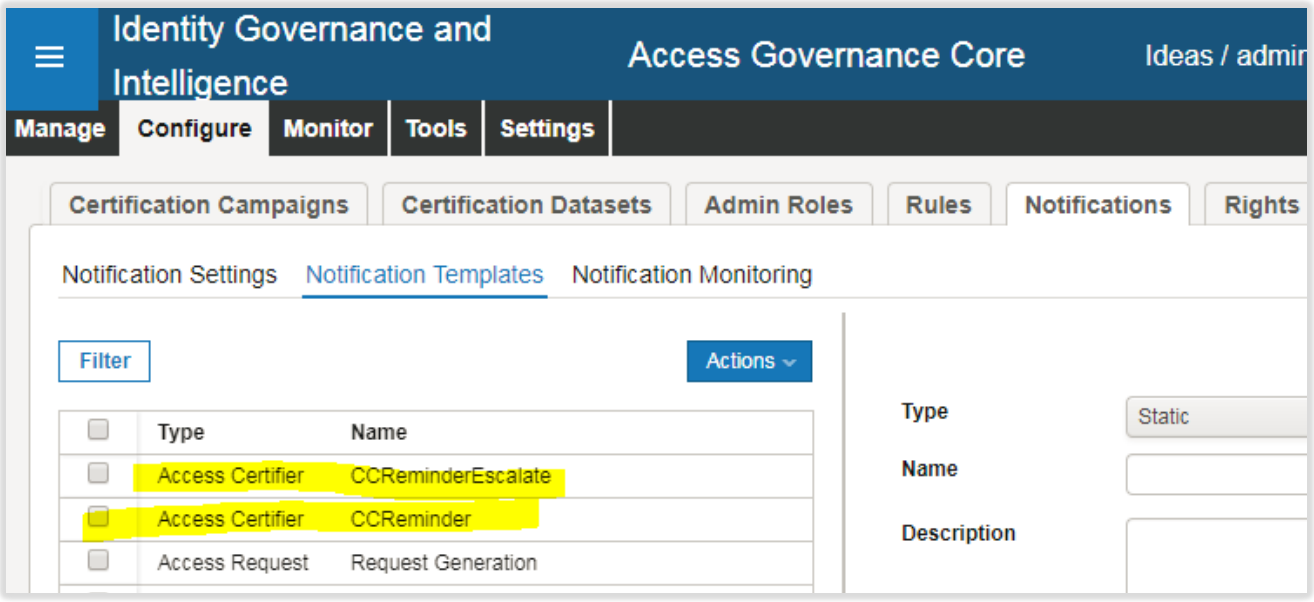
With the job defined, it is scheduled under the Scheduling tab.

Finally it must be made active to run.

6.1.4 Notification Template Configuration

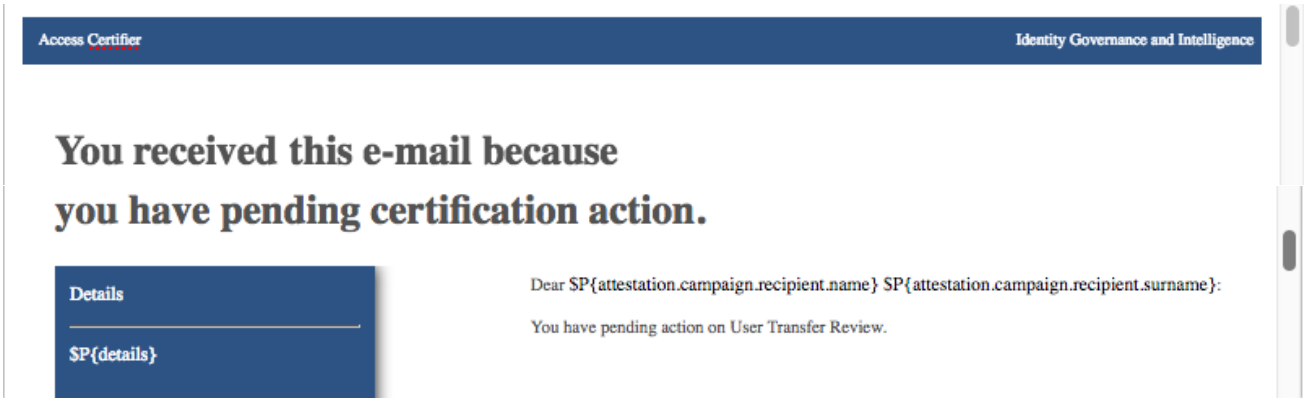
In the rules above we identified two email templates that will be used; CCReminder and CCReminderEscalate.

These are defined in the Notification Templates in Access Governance Core (Access Governance Core > Configure > Notifications > Notification Templates).



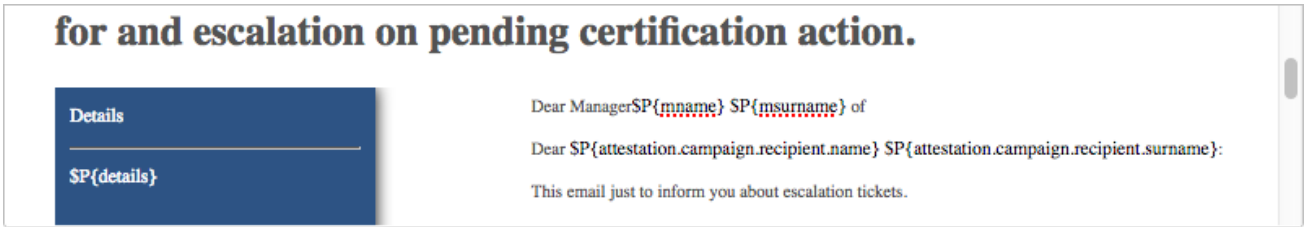
The two templates are available as samples (see directory with this document) and can be imported using the Import HTML function.

The CCReminder email template looks like this:



It includes the variables (placeholders) that are set by the rules.

The CCReminderEscalate template is similar.



Again it contains the variables (placeholders) set in the rules.

6.1.5 Testing and Executing

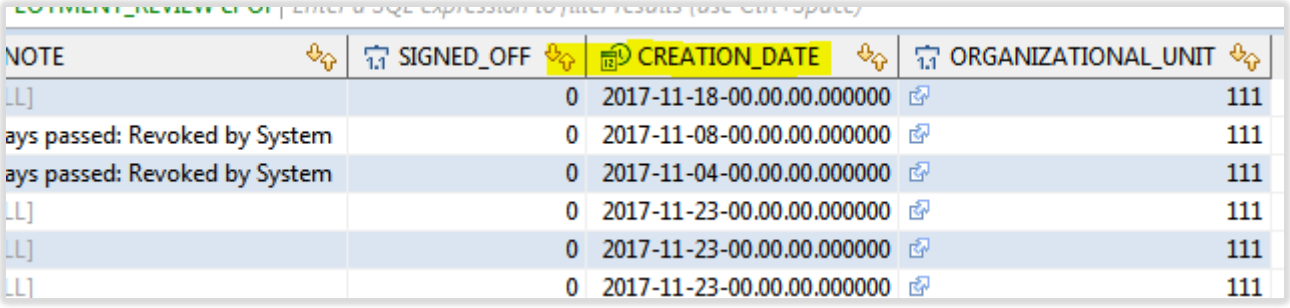
With all of the configuration above, the mechanism can be tested.

You will need a campaign with data (and a campaign name that matches the name in the rules. You cn schedule a single execution of the campaign to get it to run the rules, and depending on the logging level, you may see some messages from the rules.

However you're unlikely to see emails generated unless the dates are set. You can use your DB tool of choice to set the creation dates with the following query to see the records.

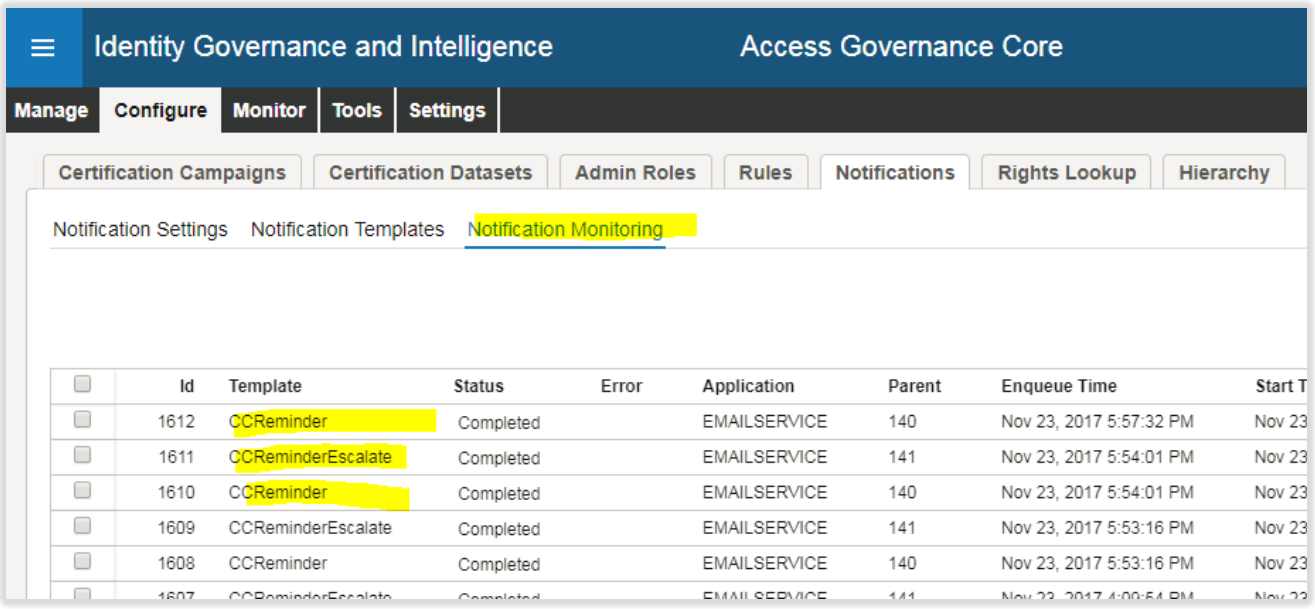
```
SELECT * FROM IGACORE.EMPLOYMENT_REVIEW er ORDER BY id DESC
```

Change the values in column CREATION_DATE (setting value in the past) to suit your testing needs.

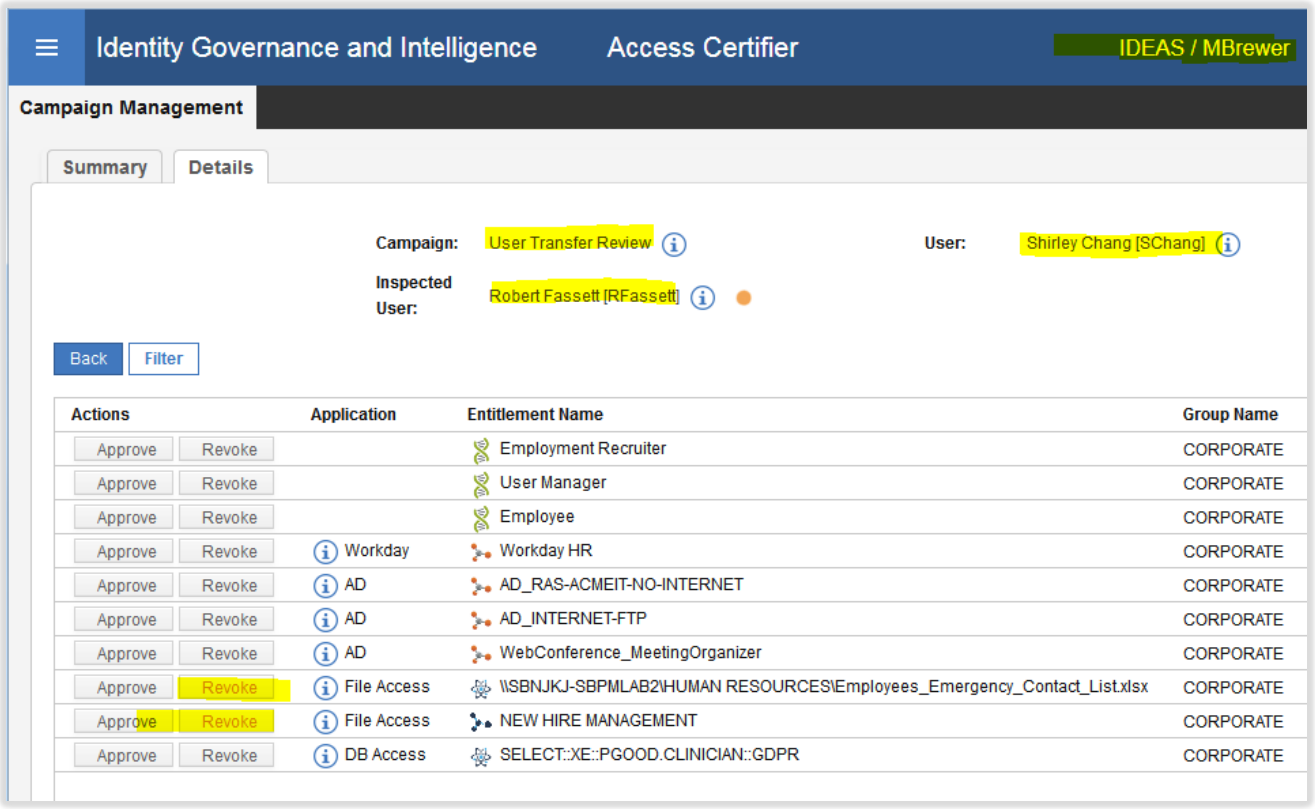


NOTE	SIGNED_OFF	CREATION_DATE	ORGANIZATIONAL_UNIT
LL]	0	2017-11-18-00.00.00.000000	111
ays passed: Revoked by System	0	2017-11-08-00.00.00.000000	111
ays passed: Revoked by System	0	2017-11-04-00.00.00.000000	111
LL]	0	2017-11-23-00.00.00.000000	111
LL]	0	2017-11-23-00.00.00.000000	111
LL]	0	2017-11-23-00.00.00.000000	111

To see the generated emails you can use the Notification Monitoring view under Notifications. This applies to production as well as testing.



Also, you can look at the campaign and see the automatically revoked entitlements.



Obviously you would set it up in production to run periodically, probably daily. If you were to run it less frequently you may need to alter the SQL in the rules to make sure you don't miss any. Also, if there is a large userbase or large number of user-entitlements in a campaign, you may want to consider when is the most appropriate time to run this.

This completes this example.

6.2 Example: Set of Rules for PoC

This example is a collection of rules for a PoC run against IGI 5.2.2. They represent some typical scenarios that needed to be implemented for a specific set of target systems. As well as some good examples, this example shows a useful approach for managing a library of rules.

6.2.1 Managing the Rules

The set of rules is held in a single Java file. This was done to ensure the rules compiled and as a means to hold them in one place. The file can be found in the attached folder and looks like this.

```
import java.sql.Timestamp;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

import com.crossideas.ideasconnector.core.databean.DataBean;
import com.crossideas.ideasconnector.core.databean.Event;
import com.engiweb.logger.impl.Log4JImpl;
import com.engiweb.pm.entity.BeanList;
import com.engiweb.pm.entity.Paging;
…
import com.engiweb.profilemanager.common.ruleengine.action.UserAction;
import com.engiweb.profilemanager.common.ruleengine.action.UtilAction;
import com.engiweb.profilemanager.common.ruleengine.action.reorganize._AccountAction;
```

```
public class IGIRules {

	private com.engiweb.logger.impl.Log4JImpl log = new Log4JImpl("");
	private com.engiweb.pm.dao.db.DAO sql = new SQLH(log);
	private com.engiweb.logger.impl.Log4JImpl logger = new Log4JImpl("");

	public void setIBMPassword() throws Exception {
		String pwd = "Passw0rd!";

		// Ideas Account
		AccountBean ideasAccountBean = new AccountBean();
		ideasAccountBean.setPwdcfg_id(1L);

		// Max 10000 Account
		Paging paging = new Paging(10000);

		BeanList<AccountBean> res = _AccountAction.findAccount(sql,
ideasAccountBean, paging);
		for (AccountBean accountBean : res) {
			_AccountAction.changePwd(sql, "", pwd, accountBean);
			System.out.println(accountBean.getEmail());
		}
	}
…
```

It contains the import commands at the top and then code to implement the objects/methods. The import commands were used in the Package Imports and the code was copied into the rules:
when
<function arguments were pasted here on separate lines>
then
<function details were pasted here as is>

The three private declarations at the top of the public class IGI Rules section are defining the logging which the Drools implementation sets up (see example rules earlier).

The rules content is in the public void sections, and we will explore some of these in later sections.

6.2.2 Package Imports

The package imports are a superset for all the classes needed for all the rules.

```
import java.sql.Timestamp;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

import com.crossideas.ideasconnector.core.databean.DataBean;
import com.crossideas.ideasconnector.core.databean.Event;
import com.engiweb.logger.impl.Log4JImpl;
import com.engiweb.pm.entity.BeanList;
import com.engiweb.pm.entity.Paging;
import com.engiweb.profilemanager.backend.dao.db.SQLH;
import com.engiweb.profilemanager.common.bean.AccountBean;
import com.engiweb.profilemanager.common.bean.Block;
import com.engiweb.profilemanager.common.bean.UserBean;
import com.engiweb.profilemanager.common.bean.UserErcBean;
import com.engiweb.profilemanager.common.bean.event.EventTargetBean;
import com.engiweb.profilemanager.common.enumerations.LockType;
import com.engiweb.profilemanager.common.ruleengine.action.UserAction;
import com.engiweb.profilemanager.common.ruleengine.action.UtilAction;
import com.engiweb.profilemanager.common.ruleengine.action.reorganize._AccountAction;
```

6.2.3 Rule Using setIdeasAccountExpiryFromUser()

This rule will get the account expiry date attribute from the current user and apply it to the Ideas account for the user. It was set on Live Events -> IN -> USER_ADD and USER_MODIFY

This has the following `when` clause:

```
when
    userBean : UserBean(  )
    userErcBean : UserErcBean(  )
    eventIn : EventInBean(  )
then
```

I've added in the eventIn bean, it's not used in the rule. The rule code is:

```
       logger.info("!!! Commencing Rule SetExpiration");
       logger.info("User to process : " + userBean.getCode());

       AccountBean userAccount = new AccountBean();
       userAccount.setPwdcfg_id(1L); // IDEAS account cfg, ID=1
       userAccount.getPwdcfg_id();
       BeanList accounts = null;
       try {
              accounts = UserAction.findAccount(sql, userBean, userAccount);
              userAccount = (AccountBean) accounts.get(0);

              Date expiryDate = (Date) userErcBean.getAttribute("ACCOUNT_EXPIRY_DATE");

              if (expiryDate != null) {
                     String stringDate = new SimpleDateFormat("dd-MM-
yyyy").format(expiryDate);
                     logger.info("Exp2:  " + stringDate);

                     if (userAccount != null) {
                            userAccount.setExpire(expiryDate);
                            logger.info("Expiration Date for Ideas set!!!!");
                            UserAction.updateAccount(sql, userAccount);
                     }
              } else {
                     logger.info("!!! No expiry date found!!! ");
              }
       } catch (Exception e) {
              logger.error("!!! Error occured in SetExpiration rule !!!");
       }
```

This rule will create new account bean ans set it's account config to be Ideas (ID = 1 Long). It will then look for the account matching that config for the user (based on the userBean passed to the rule) using the `UserAction.findAccount()` action, and saves the first account from the list (there cannot not be more than one Ideas account per user).

It then retrieves the current account expiry date from the UserErcBean (i.e. from the user record). and then finds the current account expirt date, if set. If it is set, the rule will set the same expiry date on the new account bean, and then update the actual account record with that updated bean (using the `UserAction.updateAccount();` action).

If there is no account expiry date set on the user, it logs a message and exits.

This rule could easily be extended to set the same expiry date on all user accounts.

6.2.4 Rule Using setDatesOnAccount()

This rule will set a “change password date” on a new or modified account. The user ATTR5 is being used to hold this date. It uses the account Last Change Password (LastChangePwd) and Last Login (LastLogin) attributes on the account bean. There would need to be additional processing to manage ATTR5 on the user.

This rule was set on Live Events -> TARGET -> ACCOUNT_CREATE and ACCOUNT_MODIFY

This has the following `when` clause:

```
when
    event : EventTargetBean(  )
    accountBean : AccountBean(  )
then
```

The rule code is:

```
  // Set dates on Account [ABCD]
  if (accountBean.getPerson_id() != null) {
    if (event.getTarget().equalsIgnoreCase("YYYPRD_ABCD") ||
event.getTarget().equalsIgnoreCase("ZZZPRD_ABCD")
      || event.getTarget().equalsIgnoreCase("TEST-ABCD")) {

      log.info("!!! Commencing Rule SetDatesOnAccount: " + event.getTarget());
      log.info("!!! User to process : " + accountBean.getCode() + " " +
accountBean.getName());

      // Gets the change password date
      String stringDate = event.getAttr5();
      log.info("!!! " + stringDate);
      log.info("!!! event.getAttr1()" + event.getAttr1() + "-event.getAttr5()" +
event.getAttr5())
      if (stringDate != null) {
        Date dateToSet = new SimpleDateFormat("yyyy-MM-dd").parse(stringDate);
        Timestamp timestamp = new java.sql.Timestamp(dateToSet.getTime());

        if (accountBean != null) {
          log.info("About to set Change Password Date for Account !!!!");
          accountBean.setLastChangePwd(timestamp);
          accountBean.setLastlogin(timestamp);
          log.info("Change Password Date set for Account !!!!");
          UserAction.updateAccount(sql, accountBean);
        }
      } else {
        log.info("!!! No date found to set !!! ");
      }
    }
  }
```

This rule is coded to only process account events for matched accounts (accountBean.getPErson_id() != null) beloging to any of three target systems (“YYYPRD_ABCD”, “ZZZPRD_ABCD” or “TEST-ABCD”).

If the account meets these requirements, the rule will retrieve user attr5 and format it into yyyy-MM-dd. It builds a new Timestamp object based on that date. This is used to set the LastChangePwd and LastLogin values on the account bean, and the account is updated via the `UserAction.updateAccount()` method.

6.2.5 Rule Using disableOrphanAccount()

This rule will disable orphan (unmatched) accounts.

It was set on Live Events -> TARGET -> ACCOUNT_CREATE and ACCOUNT_UNMATCHED. For the ACCOUNT_CREATE flow it would come after the account-user matching rules.

This has the following `when` clause:

```
when
    event : EventTargetBean(  )
    account : AccountBean(  )
then
```

The rule code is:

```
  if (event.getTarget().equalsIgnoreCase("YYYPRD_ABCD") ||
event.getTarget().equalsIgnoreCase("ZZZPRD_ABCD")
    || event.getTarget().equalsIgnoreCase("TEST-ABCD")) {

    Block blockCode = new Block();
    blockCode.setBlocco(0, 5);
    account.setBlock(blockCode);
    UserAction.updateAccount(sql, account);

    logger.info("Account created!");
    event.setTrace("Unable to match Identity! - Auto Disabling Account!");
    }
  }
```

This rule will be run for all account create and account unmatched events. It will only process events for accounts on YYYPRD_ABCD, ZZZPRD_ABCD and TEST-ABCD.

It will create a new Block object and set a blocked value, which is applied to the account bean and then applied to the user account via the `UserAction.updateAccount()` method.

The last line sets the trace message to the text shown. This will appear in the Trace column of the account events view (AGC > Monitor > TARGET Inbound – Account Events).

6.2.6 Rule Using matchAccount()

This rule will attempt to match an account to a user based on an email address. This is similar to one of the account matching examples shown earlier in this document.

It was set on Live Events -> TARGET -> ACCOUNT_CREATE and ACCOUNT_UNMATCHED

This has the following `when` clause:

```
when
    event : EventTargetBean(  )
    accountBean : AccountBean(  )
then
```

The rule code is:

```
    // ABCD Matching Rule(Username/Attr12 with UserId/Email)
    if (event.getTarget().equalsIgnoreCase("YYYPRD_ABCD") ||
event.getTarget().equalsIgnoreCase("ZZZPRD_ABCD")
    || event.getTarget().equalsIgnoreCase("TEST-ABCD")) {
      if (accountBean.getPerson_id() != null) {
        log.info("!!! account is already matched");
      } else {

        // Gets the email. Email Attribute stores a copy of the user id
        String email = event.getEmail();
        if (email == null) {
          log.info("!!! Empty email, matching on email not applicable... exit !!!");
        } else {

          // Look for the User into IDEAS
          UserBean userFilter = new UserBean();
          userFilter.setPhoneNumber(email);
          BeanList ul = UserAction.find(sql, userFilter);

          boolean found = !ul.isEmpty();
          if (found) {
            log.info("!!! Account Matched by phone number on person & email on account
!!!");

            // found
            userFilter = (UserBean) ul.get(0);
            log.info("!!! User Found" + userFilter.getId() + " !!!");
            accountBean.setPerson_id(userFilter.getId()); // !!

            String eventUserCode = event.getCode();
            if (eventUserCode != null) {
              accountBean.setCode(eventUserCode);
            }

            if (accountBean.getId() != null) {
              // the account already exist but it is
              // unmatched/orphan
              // Lock to Set

              UserAction.updateAccount(sql, accountBean);
              log.info("!!! Account exist but it is unmatched/orphan !!!");
            } else {
              UserAction.addAccount(sql, accountBean);
              log.info("!!! Account : " + accountBean.getCode() + " created !!!");
            }
          }
        }
      }
    }
  }
```

As with the earlier rules this one is looking for accounts for three target systems; YYYPRD_ABCD, ZZZPRD_ABCD and TEST-ABCD. It has a test at the top to check if the account is already matched, and if so it will skip the rest of the code (contained in the else clause).

If the account has not been matched, the code will get the email from the event bean, and if not null it will create a user bean, set the email address on the bean and uses it to find a matching user (with the `UserAction.find()` method).

If found it will set the account bean Person_Id to the matched user. It also sets the account code to the event user code and then it will either add or update the account using the `UserAction.addAccount()` and `UserAction.updateAccount()` methods. This allows for the rule being run in response to different events.

6.3 Example – Managing UMEs

User Multiple Entries (UMEs) are user that can be attached to users to manage multiple personas, such as a users normal accounts and access and their firecall accoints and access. The two rules shown here are for synchronising enable/disable status between the primary user and associated UMEs.

A UME entry is basically the same as a normal user entry, and can have accounts and entitlements mapped to it the same as an ordinary entry. However there are some flags on the UserBean that will be different to the normal user;

- The `UmeType` attribute (e.g. using `<userbean>.getUmeType()`) will be "UME", whereas an ordinary user will be "MASTER". These are defined by the enumerator UmeType (e.g. UmeType.MASTER).
- The `Master_code` attribute (e.g. using `<userbean>.getMaster_code()`) holds the code of the master. For example if the master user was Shirley Chang (with a code of "SChang") and there was an UME also for Shirley (with a code of "SChang1"), the UME would have a Master_code of "SChang".
- The `Master_id` attribute (e.g. using `<userbean>.getMaster_id()`) is similar but for the ID of the master user. On the master user this value equals the id of the user.

These will be used in the following examples.

Ideally these types of rule should operate off the internal queue processing user changes, but these rules were developed for the Live Events > Target > Modify queue, relying on a change to another account (e.g. AD) owned by the user.

6.3.1 Enable UME Ideas Accounts Rule

This rule will get the current block (suspend/restore) state of the Ideas account on the master user, and if it's not blocked (i.e. code = "000000") then look for all UMEs associated with that user, and for each one set the Ideas account blocked code to the same (i.e. "000000").

```
when
    userBean : UserBean(  )
then
  if (userBean.getUmeType() != UmeType.MASTER) {
    return;
  }
```

The rule will run when there is a UserBean in the event. It checks to see if the UME type is NOT "MASTER" and if so, exits the rule. We want the rule to be driven by change to the master user, not one of the linked UMEs.

```
  logger.info("Running Enable UME...");
  AccountBean masterIdeasAccount = new AccountBean();
  boolean isMasterIdeasAccountUnlocked = true;
  Block masterIdeasBlockCode = null;

  masterIdeasAccount.setPwdcfg_id(1L); // IDEAS cfg, ID=1
```

This block of code creates a new AccountBean, sets a flag for isMasterIdeasAccountUnlocked to true, and creates a new Block object. Finally, as we've seen in other rules, the new AccountBean is set to the account configuration (Pwdcfg) id of the Ideas account (1L = ideas account).

```
BeanList masterAccounts = UserAction.findAccount(sql, userBean, masterIdeasAccount);
if (!masterAccounts.isEmpty()) {
  masterIdeasAccount = (AccountBean) masterAccounts.get(0);
  masterIdeasBlockCode = masterIdeasAccount.getBlock();
  for (int i = 0; i < masterIdeasBlockCode.getStringBlocco().length; i++) {
    if (masterIdeasBlockCode.getStringBlocco()[i] != 0) {
      isMasterIdeasAccountUnlocked = false;
      logger.info("Master Ideas account locked!");
      return;
    }
  }
}
```

This empty Ideas AccountBean is used as the search argument for the account on this user, using the `UserAction.findAccount()` method.

If an Ideas account is found for the person (and it should) the rule creates as new AccountBean from the found account (there should only be one, thus the first array entry with get(0)) and determines the Block state (suspend/resume state).

It then loops through each of the block codes (there will be six) looking for a non-zero code (i.e. anything other than 000000 is considered locked). If any non-zero codes are found it sets the Unlocked flag to false, writes a log message out and exits the code.

The remaining code will only execute if the block flag is set to all zero's, i.e. the user is unlocked.

```
UserBean userFilter = new UserBean();
//Once an UME is created, the Master_id (Parent ID in the UI) is populated
userFilter.setMaster_id(userBean.getMaster_id());
```

This block of code creates a new empty UserBean and then sets the MasterID of the empty bean to that from the UserBean passed into the rule. As mentioned above, the Master_is is the id of the master user this UME is associated with, so the search is looking for all users where the master_id is set to be the id of the user driving this rule.

```
BeanList userList = UserAction.find(sql, userFilter);

if (!userList.isEmpty()) {
  for (int i = 0; i < userList.size(); i++) {
    UserBean umeUser = (UserBean)userList.get(i);
    if (umeUser.getUmeType() == UmeType.MASTER) {
      continue;
    }
    logger.info("User: "+ umeUser.getCode() +
        " found, with UME Type: "+ umeUser.getUmeType().name() +
        " and ParentID: "+ umeUser.getSwimUser() +
        " and MasterCode: "+ umeUser.getMaster_code() +
        " and MasterID: "+ umeUser.getMaster_id());
```

The code then finds the user(s) matching that master userid. This should be the master user and all associated UMEs. It scrolls through the returned list of users.

For each user it will build a UserBean and check if the user is the MASTER user, and if so it will exit the loop. Otherwise it logs information about the UME and continues processing.

```
      AccountBean userAccount = new AccountBean();
      userAccount.setPwdcfg_id(1L); // IDEAS cfg, ID=1
      logger.info("Searching for Ideas account for: "+ umeUser.getCode());
      BeanList accounts = UserAction.findAccount(sql, umeUser, userAccount);
      if (!accounts.isEmpty()) {
        logger.info("Ideas account found for user: "+ umeUser.getCode());
        userAccount = (AccountBean) accounts.get(0);

        Block blockCode = new Block();
        blockCode.setStringBlocco(masterIdeasBlockCode.getStringBlocco());

        userAccount.setBlock(blockCode);
        UserAction.updateAccount(sql, userAccount );
        logger.info("Ideas account found for user: "+ umeUser.getCode() +" enabled...");
      } else {
        logger.info("No Ideas account found for user: "+ umeUser.getCode());
      }
    }
  } else {
    logger.info("No UME found for the search criteria.");
  }
```

It will create a new AccountBean and set it to the Ideas account type.

It will look for the Ideas account for that UME user and if found it will create a new AccountBean, set a new block code based on the master account block code (will be 000000) and then update that Ideas account with the block code.

Finally, it closes the loops with some informational messages if the right conditions aren't met.

6.3.2 Disable UME Ideas Accounts Rule

The corresponding Disable UMEs rule is very similar to the Enable UMEs rule above. It will look at the block status of the Ideas account for the master user and if any of the flags are set to block (i.e. not zero) it will go through every UME associated with the user and set their Ideas account to blocked.

```
when
    userBean : UserBean(  )
then
  if (userBean.getUmeType() != UmeType.MASTER) {
    return;
  }

  logger.info("Running Disable UME...");
  AccountBean masterIdeasAccount = new AccountBean();
  boolean isMasterIdeasAccountLocked = false;
  Block masterIdeasBlockCode = null;
```

The initial block of code is basically the same as above. If the user in the UserBean triggering this rule is not a MASTER user, exit the code. It then sets some beans and variables for later use.

```
  masterIdeasAccount.setPwdcfg_id(1L); // IDEAS cfg, ID=1

  BeanList masterAccounts = UserAction.findAccount(sql, userBean, masterIdeasAccount);
```

This will setup and search for the account bean for the Ideas account for the user.

```
if (!masterAccounts.isEmpty()) {
  masterIdeasAccount = (AccountBean) masterAccounts.get(0);
  masterIdeasBlockCode = masterIdeasAccount.getBlock();
  for (int i = 0; i < masterIdeasBlockCode.getStringBlocco().length; i++) {
    if (masterIdeasBlockCode.getStringBlocco()[i] != 0) {
      isMasterIdeasAccountLocked = true;
      logger.info("Master Ideas account locked, locking child account(s)...");
      break;
    }
  }
}
```

This block of code will go through the individual block codes (six) and if any are non-zero it flags that the master ideas account is locked and will go process the other Ideas accounts for the UMEs.

```
if (!isMasterIdeasAccountLocked) {
  logger.info("Master Ideas account not locked.");
  return;
}
```

This is the exit condition if no non-zero block codes were found (i.e. the master ideas account is unlocked).

```
UserBean userFilter = new UserBean();
//Once an UME is created, the Master_id (Parent ID in the UI) is populated
userFilter.setMaster_id(userBean.getMaster_id());

BeanList userList = UserAction.find(sql, userFilter);

if (!userList.isEmpty()) {
  for (int i = 0; i < userList.size(); i++) {
    UserBean umeUser = (UserBean)userList.get(i);
    if (umeUser.getUmeType() == UmeType.MASTER) {
      continue;
    }
    logger.info("User: "+ umeUser.getCode() +
        " found, with UME Type: "+ umeUser.getUmeType().name() +
        " and ParentID: "+ umeUser.getSwimUser() +
        " and MasterCode: "+ umeUser.getMaster_code() +
        " and MasterID: "+ umeUser.getMaster_id());
```

As in the earlier example, this code is creating a new UserBean for a filter and setting that filter bean to have the Master_is as the id of the master user.

It then searches for all users where the master_id equals the id of the master user, i.e. all user records associated with the master user. This search will return the master user as well as all of the UMEs.

If the returned list is not empty (and it shouldn't be as it should return the master user as a minimum) it will scroll through all the returned user beans.

If the current user entry is a MASTER user, it skips out of the loop and iterates to the next user in the user list.

Otherwise it logs details of the UME entry.

```
      AccountBean userAccount = new AccountBean();
      userAccount.setPwdcfg_id(1L); // IDEAS cfg, ID=1
      logger.info("Searching for Ideas account for: "+ umeUser.getCode());
      BeanList accounts = UserAction.findAccount(sql, umeUser, userAccount);
      if (!accounts.isEmpty()) {
        logger.info("Ideas account found for user: "+ umeUser.getCode());
        userAccount = (AccountBean) accounts.get(0);

        Block blockCode = new Block();
        blockCode.setStringBlocco(masterIdeasBlockCode.getStringBlocco());

        userAccount.setBlock(blockCode);
        UserAction.updateAccount(sql, userAccount );
        logger.info("Ideas account found for user: "+ umeUser.getCode() +" disabled...");
      } else {
        logger.info("No Ideas account found for user: "+ umeUser.getCode());
      }
    }
  } else {
    logger.info("No UME found for the search criteria.");
  }
```

Finally, as above, it will find the Ideas account for the current UME being processed and set the block codes to match the codes on the master Ideas account.

This concludes this example.

Appendices

Appendices:
A – Working Memory Objects for Events
B – EventBean Attributes for Different Operations
C – Summary of IGI Data Flows

NOTE – the following appendices are for reference. The data was correct at the time it was produced, but later product changes may invalidate objects, methods, attributes etc. You should always test your code to verify. Hopefully this information will become official product documentation at some point and will be updated as product changes are made.

Appendix A – Working Memory Objects for Events

This appendix summarises the working memory objects available to rules (primarily the event-based rules).

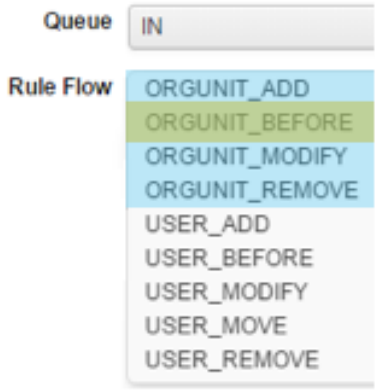
A.1 Summary of Working Memory Objects

The following table shows all the objects that may be available in working memory for event-based rules.

OBJECT	SUBOBJECT	Description
EventInBean		Event IN information
EventTargetBean		Event TARGET information
EventOutBean		Event OUT information
EventInternalBean		Event INTERNAL information
OrgUnitBean		Organization Unit information
OrgUnitErcBean		Organization Unit information in the staging area
UserBean		User information
UserErcBean		User information in the staging area
ExternalInfo		User information in the staging area mapped to the user
AccountBean		Account information
ApplicationBean		Application information
EntitlementBean		Entitlement information
RightsCnt		Right information
SyncStateBean	profile	Access Information
	syncEntitlements	List of adding/removing Entitlements that will not change the user Access assignment
	compatibleEntitlements	List of adding/removing Entitlements that will change the user Access assignment
	syncDelegateEntitlements	List of adding/removing delegated Entitlements that will not change the user Access assignment
	compatileDelegateEntitlements	List of adding/removing delegated Entitlements that will change the user Access assignment
EntStateBean	master	Access Information (parent of the entitlement hierarchy)
	profile	Access Information (child of the entitlement hierarchy)
	application	Application Information (of the parent of the entitlement hierarchy)
	childApplication	Application Information (of the child of the entitlement hierarchy)

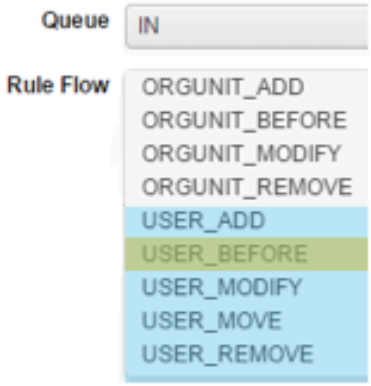
The following figures show the in-memory objects for different operations.

A.2 In-Bound (IN Queue) OU Events



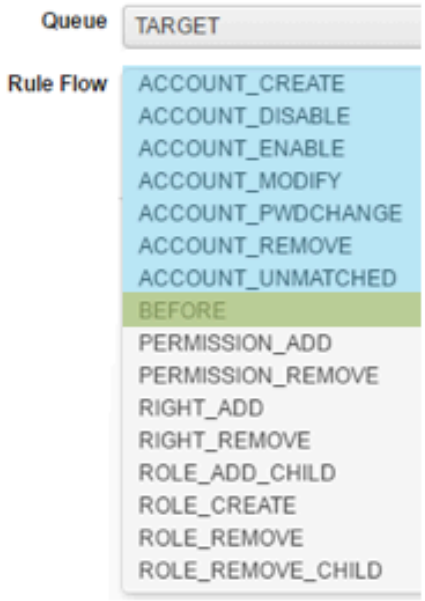
EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		IN	ORGUNIT_BEFORE	EventInBean
				OrgUnitErcBean
IN - Org. Unit	Create OU	IN	ORGUNIT_BEFORE ORGUNIT_ADD	EventInBean
				OrgUnitBean
				OrgUnitErcBean
IN - Org. Unit	Modify OU	IN	ORGUNIT_BEFORE ORGUNIT_MODIFY	EventInBean
				OrgUnitBean
				OrgUnitErcBean
IN - Org. Unit	Remove OU	IN	ORGUNIT_BEFORE ORGUNIT_REMOVE	EventInBean
				OrgUnitBean
				OrgUnitErcBean

A.3 In-Bound (IN Queue) User Events



EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		IN	USER_BEFORE	EventInBean
				UserErcBean
IN - User	Create User	IN	USER_BEFORE USER_ADD	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Modify User	IN	USER_BEFORE USER_MODIFY	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Remove User	IN	USER_BEFORE USER_REMOVE	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Move User	IN	USER_BEFORE USER_MOVE	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Custom	IN	USER_BEFORE	

A.4 In-Bound (TARGET Queue) Account Events



EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT	SUBOBJECT
		TARGET	BEFORE	EventTargetBean	
TARGET inbound - Account	Reset Password	TARGET	BEFORE ACCOUNT_PWDCHANGE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
TARGET inbound - Account	Disable User	TARGET	BEFORE ACCOUNT_DISABLE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
TARGET inbound - Account	Enable User	TARGET	BEFORE ACCOUNT_ENABLE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
TARGET inbound - Account	Unmatched User	TARGET	BEFORE ACCOUNT_UNMATCHED	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
TARGET inbound - Account	Modify User	TARGET	BEFORE ACCOUNT_MODIFY	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
TARGET inbound - Account	Create User	TARGET	BEFORE ACCOUNT_CREATE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
TARGET inbound - Account	Remove User	TARGET	BEFORE ACCOUNT_REMOVE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
				SyncStateBean	profile
					syncEntitlements
					compatibleEntitlements
					syncDelegateEntitlements
					compatileDelegateEntitlements
TARGET inbound - Account	Custom	TARGET	BEFORE		

A.5 In-Bound (TARGET Queue) Assignment (User-Access) Events

Queue	TARGET
Rule Flow	ACCOUNT_CREATE
	ACCOUNT_DISABLE
	ACCOUNT_ENABLE
	ACCOUNT_MODIFY
	ACCOUNT_PWDCHANGE
	ACCOUNT_REMOVE
	ACCOUNT_UNMATCHED
	BEFORE
	PERMISSION_ADD
	PERMISSION_REMOVE
	RIGHT_ADD
	RIGHT_REMOVE
	ROLE_ADD_CHILD
	ROLE_CREATE
	ROLE_REMOVE
	ROLE_REMOVE_CHILD

EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT	SUBOBJECT
		TARGET	BEFORE	EventTargetBean	
TARGET inbound - Account	Add Permission	TARGET	BEFORE PERMISSION_ADD	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
				SyncStateBean	profile
					syncEntitlements
					compatibleEntitlements
TARGET inbound - Account	Remove Permission	TARGET	BEFORE PERMISSION_REMOVE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
				SyncStateBean	profile
					syncEntitlements
					compatibleEntitlements
					syncDelegateEntitlements
					compatileDelegateEntitlements
TARGET inbound - Account	Add Right	TARGET	BEFORE RIGHT_ADD	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
				SyncStateBean	profile
TARGET inbound - Account	Remove Right	TARGET	BEFORE RIGHT_REMOVE	EventTargetBean	
				UserBean	
				AccountBean	
				OrgUnitBean	
				SyncStateBean	profile
TARGET inbound - Account	Custom	TARGET	BEFORE		

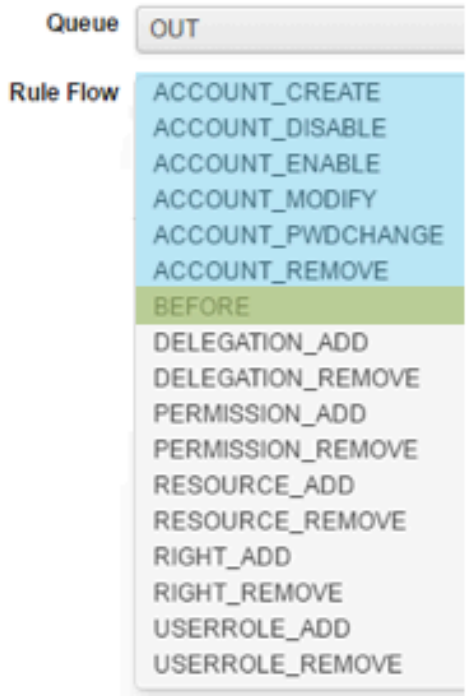
A.6 In-Bound (TARGET Queue) Access Events

Queue: TARGET

Rule Flow:
- ACCOUNT_CREATE
- ACCOUNT_DISABLE
- ACCOUNT_ENABLE
- ACCOUNT_MODIFY
- ACCOUNT_PWDCHANGE
- ACCOUNT_REMOVE
- ACCOUNT_UNMATCHED
- BEFORE
- PERMISSION_ADD
- PERMISSION_REMOVE
- RIGHT_ADD
- RIGHT_REMOVE
- ROLE_ADD_CHILD
- ROLE_CREATE
- ROLE_REMOVE
- ROLE_REMOVE_CHILD

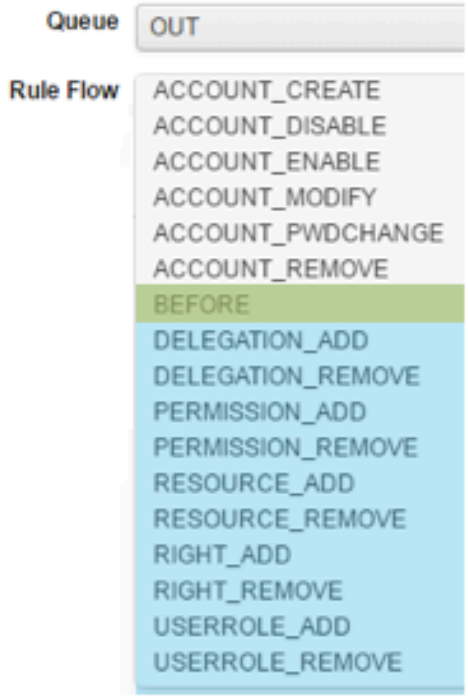
EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT	SUBOBJECT
		TARGET	BEFORE	EventTargetBean	
TARGET inbound - Account	Entitlement Add Child	TARGET	BEFORE ROLE_ADD_CHILD	EventTargetBean	
				EntStateBean	master
					profile
					application
TARGET inbound - Account	Entitlement Remove Child	TARGET	BEFORE ROLE_REMOVE_CHILD	EventTargetBean	
				EntStateBean	master
					profile
					application
TARGET inbound - Access	Create External Role	TARGET	BEFORE ROLE_CREATE	EventTargetBean	
				RightsCnt	
				EntStateBean	master
					profile
					application
					childApplication
TARGET inbound - Access	Delete External Role	TARGET	BEFORE ROLE_REMOVE	EventTargetBean	
				EntStateBean	master
					profile
					application
					childApplication
TARGET inbound - Access	Add External Role Child	TARGET	BEFORE ROLE_ADD_CHILD	EventTargetBean	
				EntStateBean	master
					profile
					application
					childApplication
TARGET inbound - Access	Remove External Role Child	TARGET	BEFORE ROLE_REMOVE_CHILD	EventTargetBean	
				EntStateBean	master
					profile
					application
					childApplication

A.7 Out-Bound (OUT Queue) Account Events

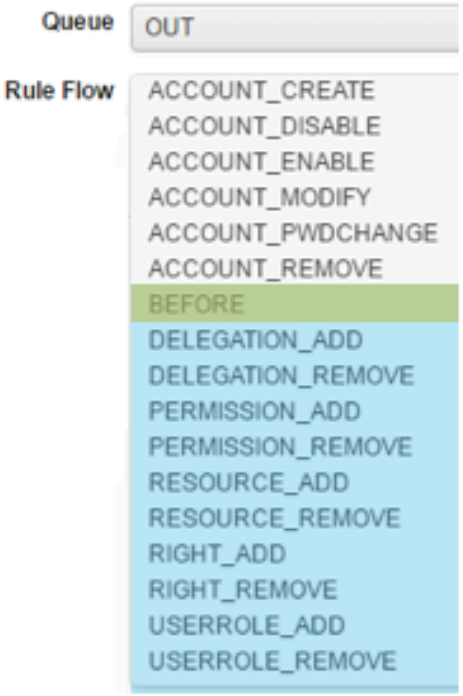


EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		OUT	BEFORE	EventOutBean
OUT	Disable User	OUT	BEFORE ACCOUNT_DISABLE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Enable User	OUT	BEFORE ACCOUNT_ENABLE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Create Account	OUT	BEFORE ACCOUNT_CREATE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Remove Account	OUT	BEFORE ACCOUNT_REMOVE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Modify Account	OUT	BEFORE ACCOUNT_MODIFY	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Change Password	OUT	BEFORE ACCOUNT_PWDCHANGE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo

A.8 Out-Bound (OUT Queue) Assignment (User-Access) Events

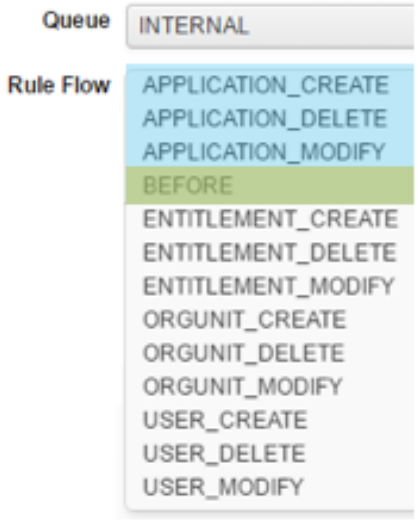


EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		OUT	BEFORE	EventOutBean
OUT	Add Permission	OUT	BEFORE PERMISSION_ADD	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Remove Permission	OUT	BEFORE PERMISSION_REMOVE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Add Delegation	OUT	BEFORE DELEGATION_ADD	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Remove Delegation	OUT	BEFORE DELEGATION_REMOVE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Add Right	OUT	BEFORE RIGHT_ADD	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Remove Right	OUT	BEFORE RIGHT_REMOVE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo



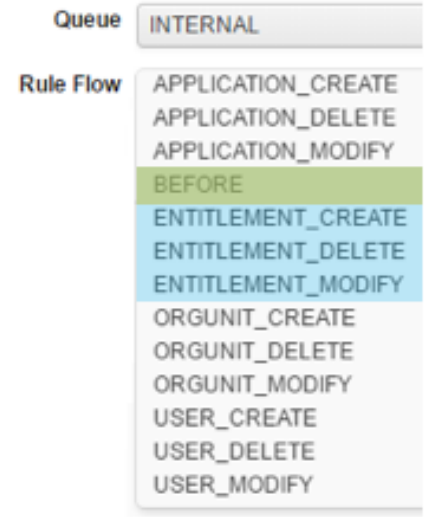
EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		OUT	BEFORE	EventOutBean
OUT	Add Resource	OUT	BEFORE RESOURCE_ADD	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Remove Resource	OUT	BEFORE RESOURCE_REMOVE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Adding Entitlement to User	OUT	BEFORE USERROLE_ADD	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo
OUT	Removal Entitlement to User	OUT	BEFORE USERROLE_REMOVE	EventOutBean
				UserBean
				OrgUnitBean
				AccountBean
				ExternalInfo

A.9 Out-Bound (INTERNAL Queue) Application Events



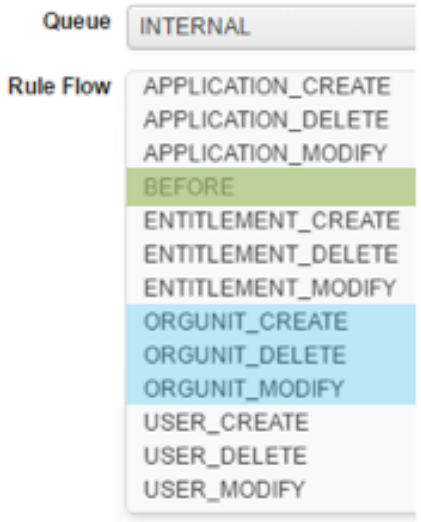
EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		INTERNAL	BEFORE	EventInternalBean
INTERNAL	Create Application	INTERNAL	BEFORE APPLICATION_CREATE	EventInternalBean ApplicationBean
INTERNAL	Remove Application	INTERNAL	BEFORE APPLICATION_DELETE	EventInternalBean
INTERNAL	Modify Application	INTERNAL	BEFORE APPLICATION_MODIFY	EventInternalBean ApplicationBean

A.10 Out-Bound (INTERNAL Queue) Entitlement Events



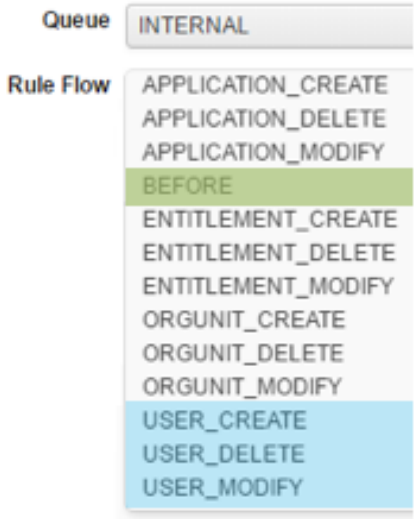
EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		INTERNAL	BEFORE	EventInternalBean
INTERNAL	Create Entitlement	INTERNAL	BEFORE ENTITLEMENT_CREATE	EventInternalBean EntitlementBean
INTERNAL	Remove Entitlement	INTERNAL	BEFORE ENTITLEMENT_DELETE	EventInternalBean
INTERNAL	Modify Entitlement	INTERNAL	BEFORE ENTITLEMENT_MODIFY	EventInternalBean EntitlementBean

A.11 Out-Bound (INTERNAL Queue) OU Events



EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		INTERNAL	BEFORE	EventInternalBean
INTERNAL	Create OU	INTERNAL	BEFORE ORGUNIT_CREATE	EventInternalBean OrgUnitBean
INTERNAL	Remove OU	INTERNAL	BEFORE ORGUNIT_DELETE	EventInternalBean
INTERNAL	Modify OU	INTERNAL	BEFORE ORGUNIT_MODIFY	EventInternalBean OrgUnitBean

A.12 Out-Bound (INTERNAL Queue) User Events



EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		INTERNAL	BEFORE	EventInternalBean
INTERNAL	Create User	INTERNAL	BEFORE USER_CREATE	EventInternalBean UserBean ExternalInfo
INTERNAL	Remove User	INTERNAL	BEFORE USER_DELETE	EventInternalBean
INTERNAL	Modify User	INTERNAL	BEFORE USER_MODIFY	EventInternalBean UserBean ExternalInfo

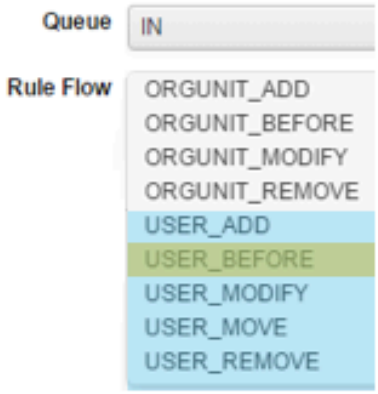
A.13 In-Bound (IN Queue) OU Events – DEFERRED

Queue: IN

Rule Flow:
- ORGUNIT_ADD
- ORGUNIT_BEFORE
- ORGUNIT_MODIFY
- ORGUNIT_REMOVE
- USER_ADD
- USER_BEFORE
- USER_MODIFY
- USER_MOVE
- USER_REMOVE

EVENT	OPERATION	CLASS	QUEUE	RULE FLOW	OBJECT
		Deferred Events	IN	ORGUNIT_BEFORE	EventInBean OrgUnitErcBean
NOT YET IMPLEMENTED	NOT YET IMPLEMENTED	Deferred Events	IN	ORGUNIT_BEFORE ORGUNIT_ADD	
NOT YET IMPLEMENTED	NOT YET IMPLEMENTED	Deferred Events	IN	ORGUNIT_BEFORE ORGUNIT_MODIFY	
NOT YET IMPLEMENTED	NOT YET IMPLEMENTED	Deferred Events	IN	ORGUNIT_BEFORE ORGUNIT_REMOVE	

A.14 In-Bound (IN Queue) User Events – DEFERRED



EVENT	OPERATION	QUEUE	RULE FLOW	OBJECT
		IN	ORGUNIT_BEFORE	EventInBean
				OrgUnitErcBean
IN - User	Create User (Deferral)	IN	USER_BEFORE USER_ADD	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Modify User (Deferral)	IN	USER_BEFORE USER_MODIFY	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Remove User (Deferral)	IN	USER_BEFORE USER_REMOVE	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean
IN - User	Move User (Deferral)	IN	USER_BEFORE USER_MOVE	EventInBean
				UserErcBean
				UserBean
				ExternalInfo
				OrgUnitBean

Appendix B – EventBean Attributes for Different Operations

The following attribute information is not published and may contain version-specific information. Hopefully it will form part of the official documentation at some point. Until then, please treat the information with care and test.

B.1 EVENT_TARGET Events

B.1.1 Account Events

EVENT_TARGET	Description	NOTE
OPERATION	**10 Account Create**	
	9 Account Modify	
	11 Account Delete	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
CODE	Account User ID	required
NAME	User's personal data imported from target account	optional
SURNAME	User's personal data imported from target account	optional
DN	User's personal data imported from target account	optional
DISPLAY_NAME	User's personal data imported from target account	optional
EMAIL	User's personal data imported from target account	optional
IDENTITY_UID	Identity unique identifier	optional - if available it can be used to match id
ATTR1	not used	
ATTR2	Encrypted password	optional
ATTR3	EXPIRE_DATE	optional - System date format
ATTR4	LAST_ACCESS_DATE	optional - System date format
ATTR5	LAST_WRONG_LOGIN	optional
ATTR6	NUMBER_LOGIN_ERROR	optional
ATTR7	LAST_PWD_CHANGE	optional
ATTR8	not used	
ATTR9	not used	
ATTR10-20	free for custom attributes	optional

EVENT_TARGET	Description	NOTE
OPERATION	**6 Account lock**	
	7 Account unlock	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
CODE	Account User ID	required

EVENT_TARGET	Description	NOTE
OPERATION	**3 Reset Password for the Account**	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
CODE	Account User ID	required
ATTR1	not used	
ATTR2	Encrypted password	required

B.1.2 Authorization Events

EVENT_TARGET	Description	NOTE
OPERATION	**1 add Permission to a user**	
	2 remove Permission from a user	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
PROCESS_ID	identifier of the process that generated an event or group	optional
CODE	Account User ID	required
FUNCTIONALITY	Entitlement name	required if ATTR3 is null
ATTR1	not used	
ATTR2	not used	
ATTR3	Entitlement name external ref	required if FUNCTIONALITY is null
ATTR4	Entitlement type - [PERMISSION/EXTERNAL_ROLE] or [1/2]	optional - default is PERMISSION
ATTR5	Entitlement Description	optional - used only for ent. creation when it does not exist
APPLICATION	Application	optional
FUNCTIONALITY_TYPE	Permission type	

EVENT_TARGET	Description	NOTE
OPERATION	**12 add Right to a user**	
	13 remove Right to a user	
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
CODE	Account User ID	required
FUNCTIONALITY	Entitlement name	required if ATTR3 is null
ATTR1	not used	
ATTR2	not used	
ATTR3	Entitlement name external ref	required if FUNCTIONALITY is null
ATTR4	Entitlement type - [PERMISSION/EXTERNAL_ROLE] or [1/2]	optional - default is PERMISSION
ATTR5	Right name	required
ATTR6	Right value	required
APPLICATION	Application	optional
FUNCTIONALITY_TYPE	Permission type	optional

B.1.3 Entitlements Catalog Events

EVENT_TARGET	Description	NOTE
OPERATION	**25 Create External role / Permission**	
	26 Remove External role / permission	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
CODE	not used	required on the DB
ATTR1	Entitlement Name	required if ATTR3 is null
ATTR2	Permission type	
ATTR3	Entitlement name external ref	required if ATTR1 is null
ATTR4	Entitlement type - [PERMISSION/EXTERNAL_ROLE] or [1/2]	optional - default is PERMISSION
ATTR5	Description	optional
ATTR6	1 = has rights	optional
APPLICATION	Application	optional

EVENT_TARGET	Description	NOTE
OPERATION	**27 Add child**	
	28 Remove Child	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	required
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
CODE	not used	required on the DB
ATTR1	PARENT Entitlement Name	required if ATTR3 is null
ATTR2	PARENT Permission type	
ATTR3	PARENT Entitlement name external ref	required if ATTR1 is null
ATTR4	PARENT Entitlement type - [PERMISSION/EXTERNAL_ROLE] or [1/2]	required - must be EXTERNAL_ROLE
ATTR5	PARENT entitlement Description	optional
ATTR6	Entitlement Name	required if ATTR8 is null
ATTR7	Permission type	
ATTR8	Entitlement name external ref	required if ATTR6 is null
ATTR9	Entitlement type - [PERMISSION/EXTERNAL_ROLE] or [1/2]	optional - default is EXT_ROLE
ATTR10	Description	optional
ATTR11	Event marker for child	
APPLICATION	Application	optional

B.2 OUT / USER_EVENT-ERC Events

B.2.1 Account Events

USER_EVENT_ERC	Description	NOTE
OPERATION	**8 Account Create**	
	10 Account Modify	
	9 Account Delete	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	
PERSON	FK to PERSON table (Identity master data)	
USER_ERC	FK to USER_ERC table (Identity extended data)	
CODE	Account User ID	
ATTR2	Encrypted password	
ATTR3	Lock code - six digit from 0 to 9 representing IGI lock flags	active = 000000
EVENT_OUT.CHANHELOG	FK to table CHANGELOG containing all account attributes changed	since 5.2.3

USER_EVENT_ERC	Description	NOTE
OPERATION	**6 Account lock**	
	7 Account unlock	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	
PERSON	FK to PERSON table (Identity master data)	
USER_ERC	FK to USER_ERC table (Identity extended data)	
CODE	Account User ID	
ATTR3	Lock code - six digit from 0 to 9 representing IGI lock flags	

USER_EVENT_ERC	Description	NOTE
OPERATION	**11 Reset Password for the Account**	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	
PERSON	FK to PERSON table (Identity master data)	
USER_ERC	FK to USER_ERC table (Identity extended data)	
CODE	Account User ID	
ATTR2	Encrypted password	

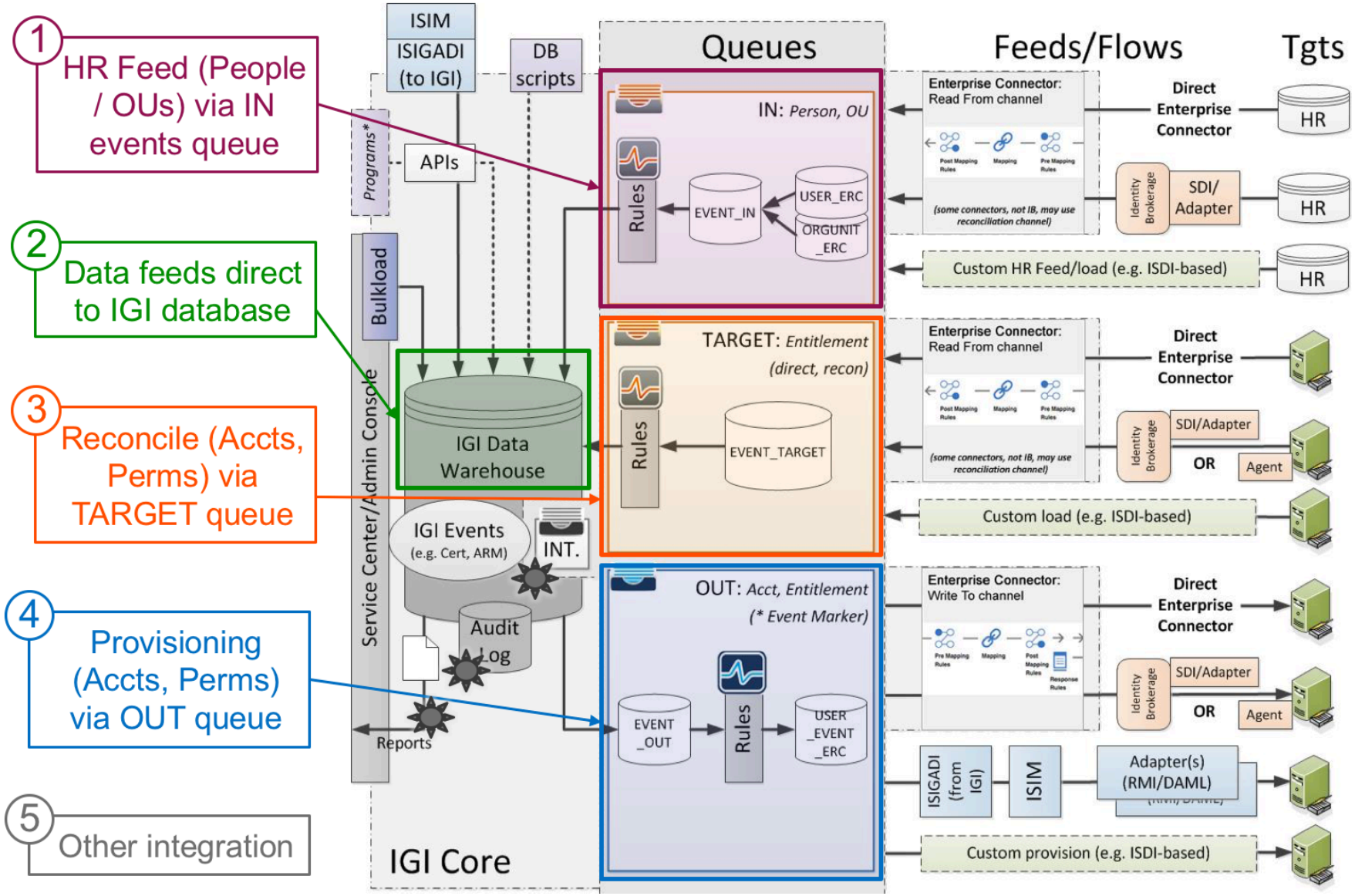
B.2.2 Authorization Events

USER_EVENT_ERC	Description	NOTE
OPERATION	**1 add Permission to a user**	
	2 remove Permission to a user	
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	
PERSON	FK to PERSON table (Identity master data)	
USER_ERC	FK to USER_ERC table (Identity extended data)	
CODE	Account User ID	
APPLICATION	Application	
ATTR1	Entitlement name	
ATTR2	Permission type	
ATTR3	Entitlement External ref	
ATTR4	1 – No Group Permission	
ATTR5	Entitlement type	

USER_EVENT_ERC	Description	NOTE
OPERATION	**12 add Right to a user**	
	13 remove Right to a user	
PROCESS_ID	identifier of the process that generated an event or a group of events	optional
TARGET	Event Marker = Target name = ILC_globalID for IGI 5.2.1/2	
PERSON	FK to PERSON table (Identity master data)	
USER_ERC	FK to USER_ERC table (Identity extended data)	
CODE	Account User ID	
APPLICATION	Application	
ATTR1	right name	
ATTR2	right value	
ATTR3	Entitlement External ref	
ATTR4	1 – No Group Permission	
ATTR5	Entitlement type	
Note:	*Missing entitlement name and permission type*	

Appendix C – Summary of IGI Data Flows

The following figure shows the IGI data flows (NOTE – technically the rules/rules engine is not part of the queues, they operate on the queues).



End of Document

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law :

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

If you are viewing this information in softcopy form, the photographs and color illustrations might not be displayed.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at ibm.com/legal/copytrade.shtml.

Statement of Good Security Practices

IT system security involves protecting systems and information through prevention, detection and response to improper access from within and outside your enterprise. Improper access can result in information being altered, destroyed, misappropriated or misused or can result in damage to or misuse of your systems, including for use in attacks on others. No IT system or product should be considered completely secure and no single product, service or security measure can be completely effective in preventing improper use or access. IBM systems, products and services are designed to be part of a comprehensive security approach, which will necessarily involve additional operational procedures, and may require other systems, products or services to be most effective. IBM DOES NOT WARRANT THAT ANY SYSTEMS, PRODUCTS OR SERVICES ARE IMMUNE FROM, OR WILL MAKE YOUR ENTERPRISE IMMUNE FROM, THE MALICIOUS OR ILLEGAL CONDUCT OF ANY PARTY.

International Business Machines Corporation
New Orchard Road Armonk, NY 10504
Produced in the United States of America 01-2016