Flux

Rule-based Authorisation for WebGUI

Version 1.2 - Patrick Donelan (2008-07-02)

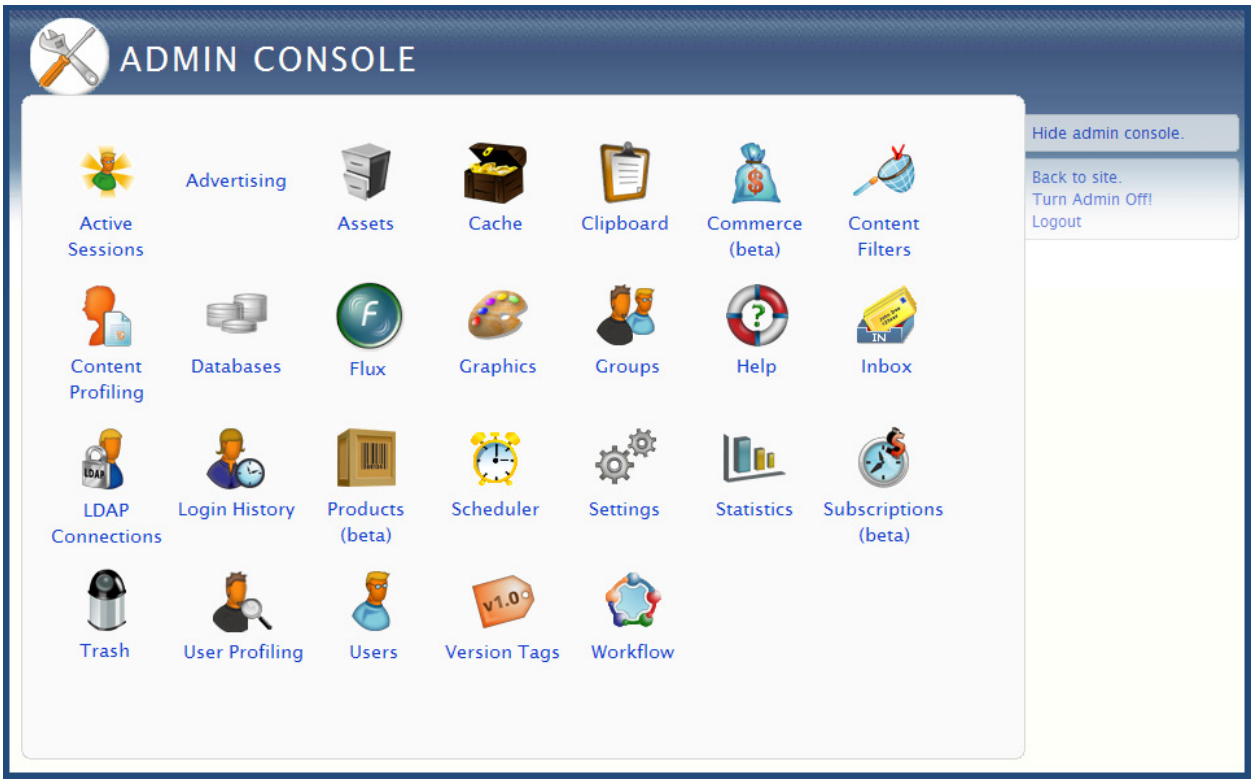
Introduction

Flux adds a dynamic behavioral layer on top of wG, giving content managers a simple and yet immensely powerful way to add rule-based authorisation to their websites. The design aims for a system that is flexible and extensible to developers but simple and intuitive to content managers.

The basic premise of Flux is incredibly simple. Currently content managers choose a group that users must belong to for them to be allowed to perform any action that the Wobject supports (View/Edit/etc..). I propose extending this mechanism so that instead of choosing a group, content managers can choose a **Flux Rule** that must evaluate to true for a user to be granted access.

Flux Rules are comprised of one or more Boolean **Expressions**. Rules and Expressions are manipulated by content managers through a simple graphical interface. The power of Flux lies in the fact that Rules can be based on user-specific, date-specific and/or Wobject-specific information. Rules can also depend on other Rules, meaning that we end up with a **Flux Graph** of interconnected Rules. Workflow triggers are built-in, and Flux is designed to be modular with many plug-in points, making the system truly extensible.

Flux adds a single item to the Admin Console called, unsurprisingly, "Flux". Flux is entirely optional - content managers can ignore this Admin Console entry entirely and happily use wG without it.



N.B. All screenshots are for reference only. Visual styling is subject to change. YUI will be used throughout for Ajax (not ExtJS as suggested by the screenshots).

The approach is straight-forward: content managers first create Rules in the Admin Console, and then use the Edit Wobject page on individual Wobjects to set "Who Can View", "Who Can Edit" to a specific Rule. The power of the framework is that access is now based on dynamic Rules rather than hardwired groups.

Flux Rules

When content managers click on the "Flux" entry in the Admin Console, they are shown the list of **Rules** that have been created for their website. This list is initially empty.

A table called **fluxRule** will holds information about all Flux Rules that have been created. It looks like this:

- fluxRuleId - primary key
- name :string - a human readable name
- sequenceNumber - defines the display order of the Flux Rule
- sticky :bool - whether the Rule needs to be re-computed every time a user requests access or whether it becomes permanent ("sticky") for them the first time access is granted
- onRuleFirstTrueWorkflowId - a Workflow to run the first time Flux notices that this Rule evaluates to true for a given user
- onRuleFirstFalseWorkflowId - a Workflow to run the first time Flux notices that this Rule evaluates to false for a given user
- onAccessFirstTrueWorkflowId - a Workflow to run the first time a user successfully attempts to perform an action that is controlled by this Rule (e.g. when a user successfully tries to View a Wobject where "Who Can View" is set to this Rule)
- onAccessFirstFalseWorkflowId - a Workflow to run the first time a user unsuccessfully attempts to perform an action that is controlled by this Rule (e.g. when a user unsuccessfully tries to View a Wobject where "Who Can View" is set to this Rule)
- onAccessTrueWorkflowId - same as onAccessFirstTrueWorkflowId but runs on every time
- onAccessFalseWorkflowId - same as onAccessFirstFalseWorkflowId but runs on every time
- combinedExpression - an initially empty string, records the way in which multiple Expressions are combined to form a single Rule in the form of a general Boolean expression, e.g. "E1 AND (E2 OR E3)" (explained in more detail later)

A second table called **fluxRuleUserData** stores individual user data relating to each Flux Rule, one entry per Rule/User combination (entries only added the first time a Rule/User combination is encountered) . It looks like this:

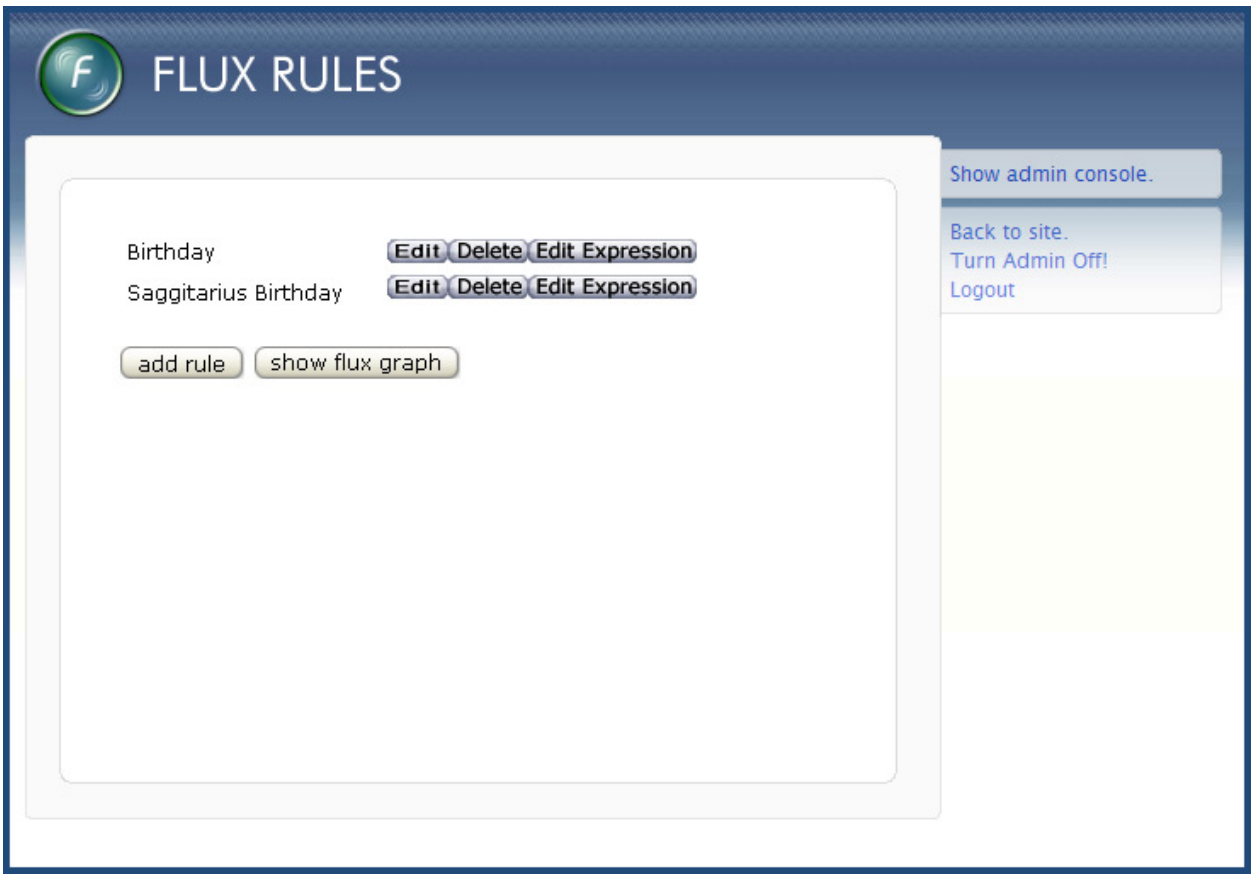
- fluxRuleUserDataId - primary key
- fluxRuleId - foreign key reference to the id of the fluxRule table
- userId - id of the user that this row refers to
- dateRuleFirstChecked :date - set to now() the first time Flux evaluates this Rule for the user
- dateRuleFirstTrue :date - set to now() the first time this Rule evaluates true for the user
- dateRuleFirstFalse :date - set to now() the first time this Rule evaluates false for the user
- dateAccessFirstAttempted :date - set to now() the first time a user attempts to perform an action that is controlled by this Rule
- dateAccessFirstTrue :date - set to now() the first time a user successfully attempts to perform an action that is controlled by this Rule
- dateAccessFirstFalse :date - set to now() the first time a user unsuccessfully attempts to perform an action that is controlled by this Rule
- dateAccessMostRecentlyTrue :date - same as dateAccessFirstTrue but updated every time*

- dateAccessMostRecentlyFalse :date - same as dateAccessFirstFalse but updated every time*

* These are included for maximum flexibility, however if they cause too much of a performance hit they should either be scrapped or made optional (performance optimisations are discussed in more detail in the Flux Graph section).

We distinguish between "checking" a rules and "accessing" a Rule so that content manager can have one workflow performed when Flux first notices that a date-based Rule evaluates to true for a user (i.e. so that you can send them an email saying "Visit our site and check out your new content!") and another performed when users actually access the content. Flux may "notice" that a Rule becomes true/false as a side-effect of checking access on a Rule that depends on other Rules, or via a special scheduled Workflow that runs periodically to check for changes in date-based Rules.

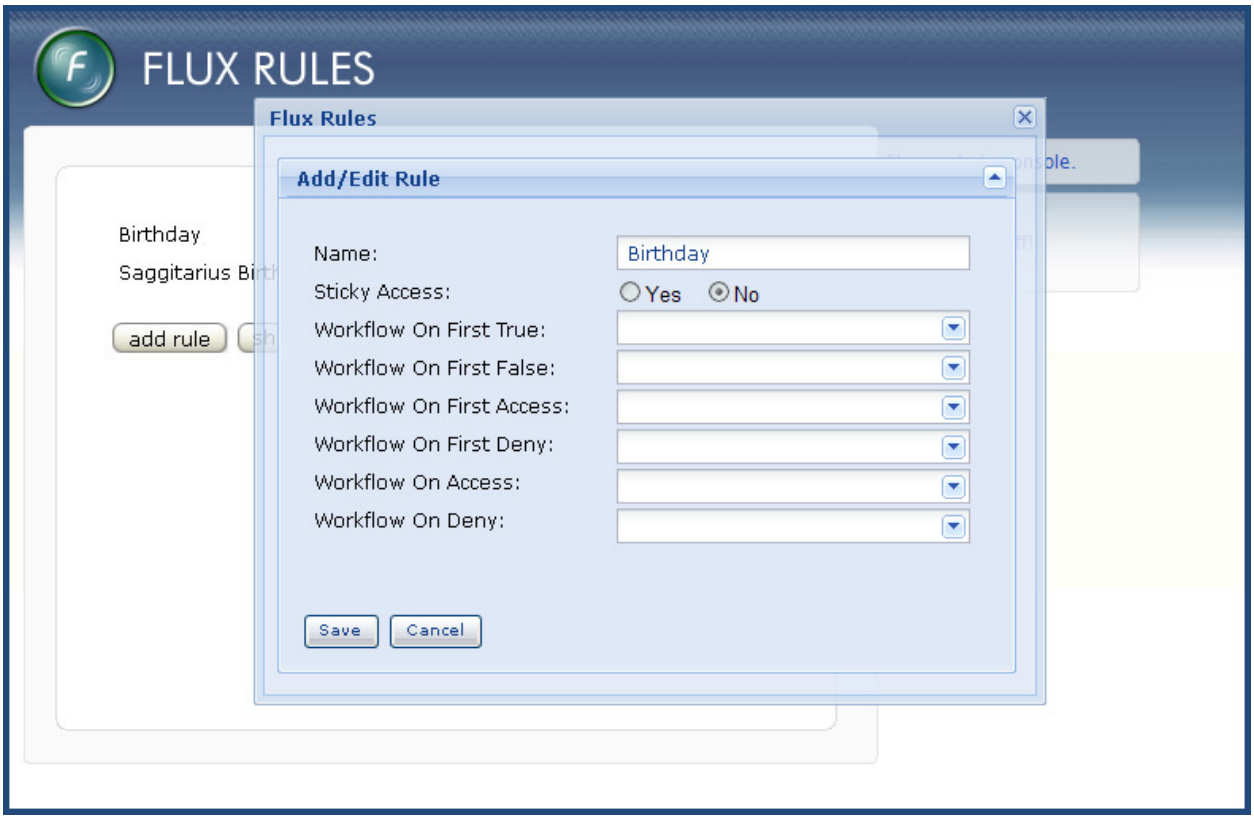
The list of Flux Rules will appear as follows:



Content managers can:

- Add Rule - prompts user for: label, sticky and Workflow fields (as per fluxRule table definition)
- Delete Rule
- Edit Rule - lets user edit all fields given above in "Add Rule"
- Edit Expressions - lets the user edit the Expressions that the selected Rule contains
- Show Graph - displays the Flux Graph (described in the Flux Graph section)

The Rule Add/Edit screen will appear as follows:



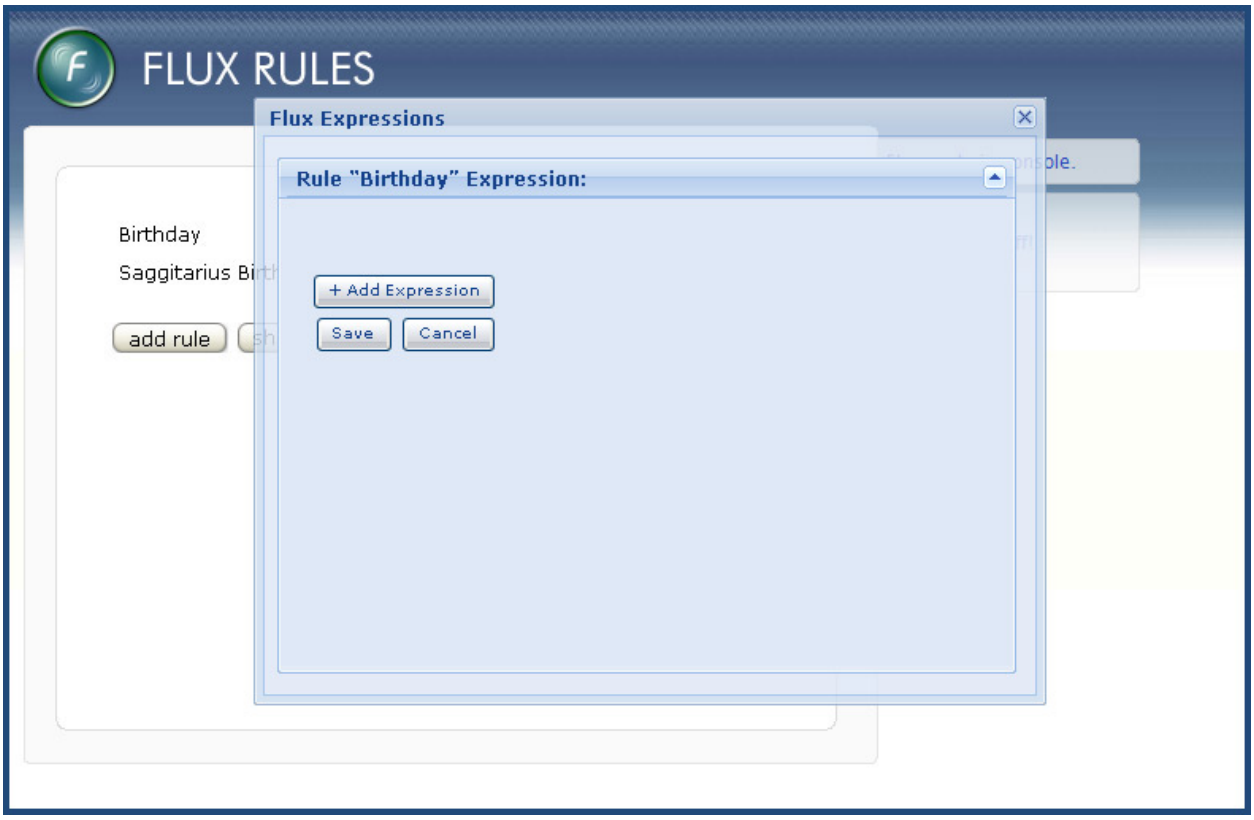
Expressions

Each Flux Rule is made up of one or more **Expressions**. In turn, each Expression is comprised of a Boolean infix **Operator** and two **Operands**, e.g.

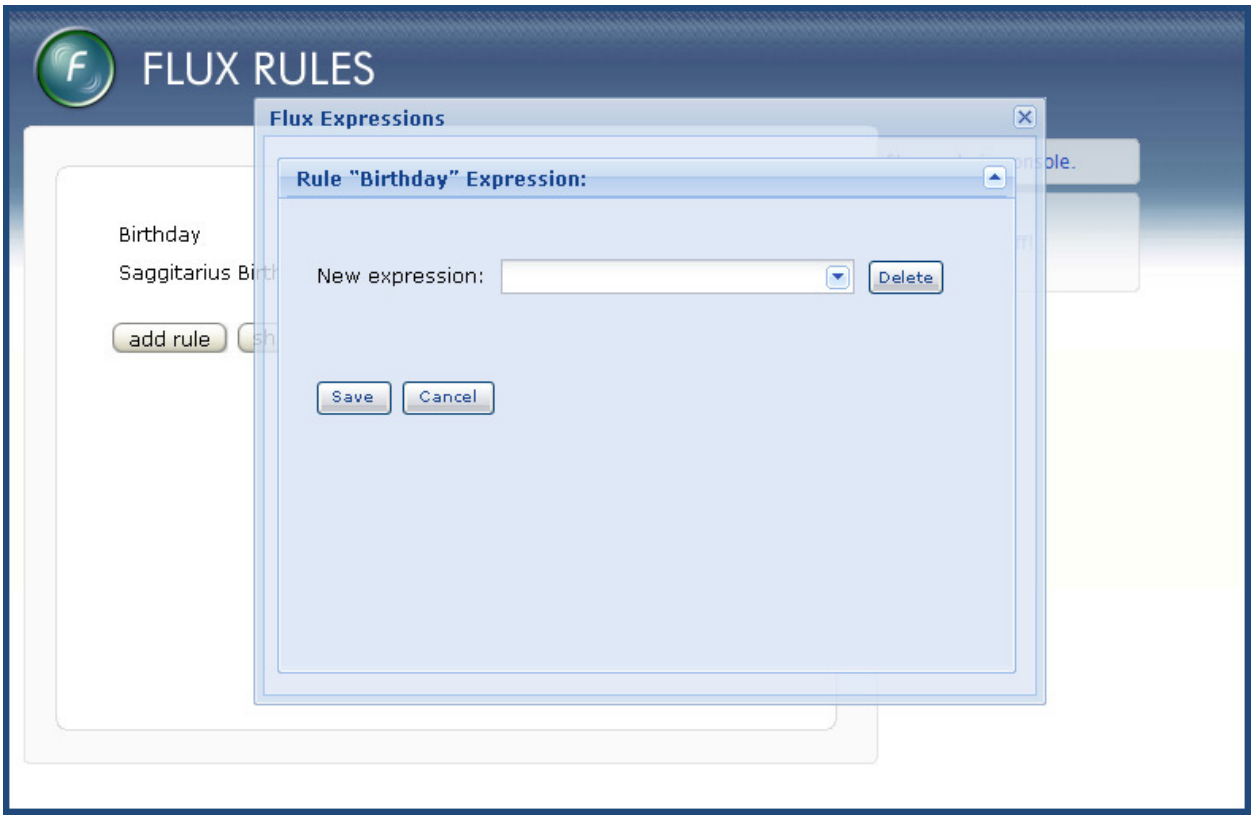
Rule:
Expression 1:
<Operand 1> <Operator> <Operand 2>
Expression 2:
<Operand 1> <Operator> <Operand 2>
Expression 3:
<Operand 1> <Operator> <Operand 2>

Obviously concepts such as Operands and Operators are programmer-friendly and not content manager-friendly. Therefore, natural language words are used wherever possible in the user interface. Also, we try to hide the inherit complexity of the model behind an intuitive user interface that only shows items when they are relevant.

Flux Rules are initially empty:

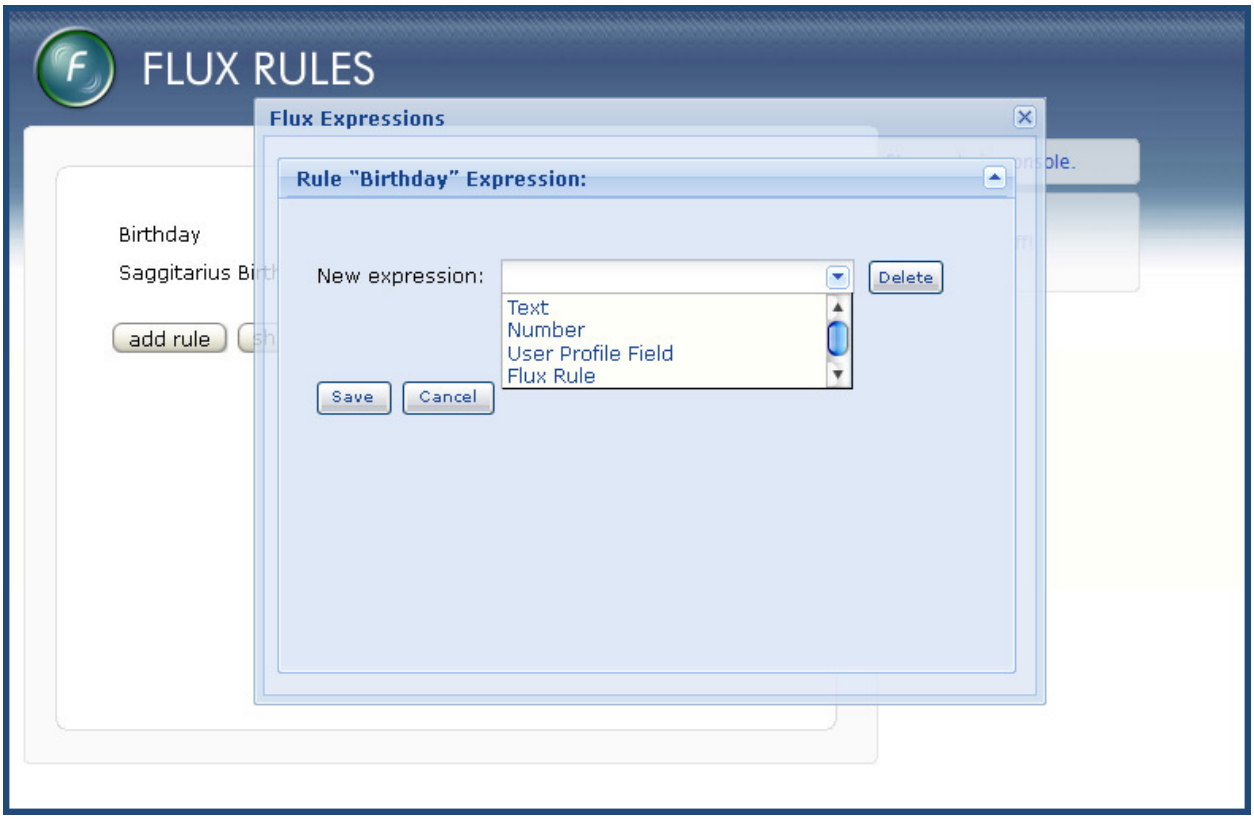


Content managers begin adding an Expression to a Rule by clicking on the "Add Expression" button. They are immediately presented with the **Operand combo box**, a dynamically populated combo box which they use to choose the first Operand in their Expression.



WebGUI::Flux::Operand

The Operand combo box is dynamically populated with all items that can be used to construct an Operand:



Each item in the combo box corresponds to a Perl module in the **WebGUI::Flux::Operand** namespace. WebGUI::Flux::Operand modules define:

- what name they should appear as in the Operand combo box (developers are encouraged to use appropriate names so that Expressions constructed with them read as naturally as possible for non-technical users)
- what extra information (user input fields) the content manager should be prompted for when they are selected
- what dynamic help/tooltips should be displayed when they are selected
- what type of value the Operand will evaluate to (String, DateTime etc..)
- whether they are bound to a specific Wobject type
- an optional list of values that the user can select from if they ask for the "suggested" list

Normally the relationship between WebGUI::Flux::Operand modules and items in the Operand combo box is one-to-one. However any items that are bound to a specific Wobject type appear as one item per Wobject instance, allowing the WebGUI::Flux::Operand to contain behavior specific to a single Wobject instance.

WebGUI::Flux::Operand modules have an execute() subroutine that is called when they are used in a Rule. This method has access to:

- The user for whom the Rule is being evaluated against
- The Rule being evaluated
- The values of any user input fields they prompted the content manager for

- The assetId of the Wobject that the user is attempting to access (or undefined if the user is not attempting to access a Wobject)
- The assetId of the Wobject instance that the WebGUI::Flux::Operand is bound to (or undefined if it is not bound to a Wobject)

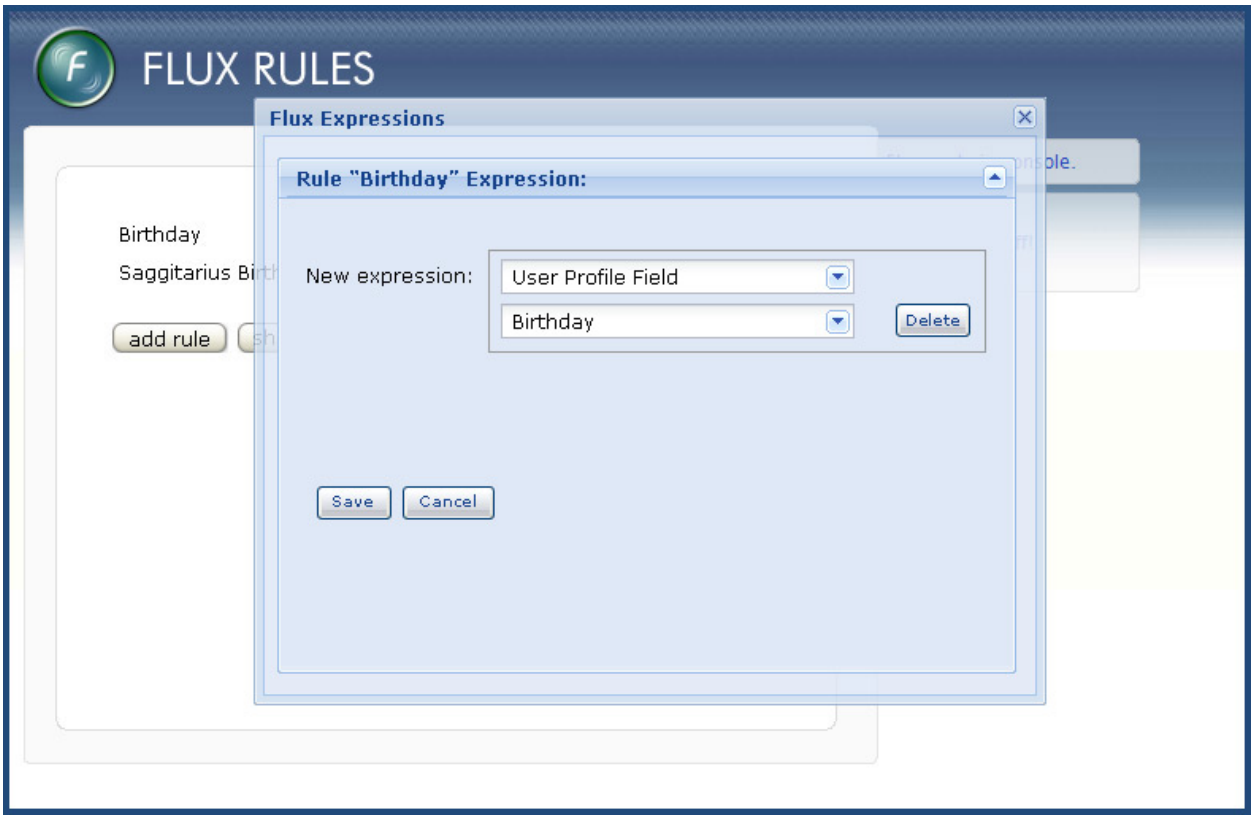
Flux will ship with a number of WebGUI::Flux::Operand modules by default, and developers are encouraged to add their own to extend the framework. The wG site config file will be used to control which Operands are enabled (via the same mechanism used for Macros).

The following WebGUI::Flux::Operand modules will be enabled by default:

- "Text Value" - when selected, a single WebGUI::Form::Text field appears into which a text value can be entered
- "Numerical Value" - when selected, a single WebGUI::Form::Number input field appears into which a numeric value can be entered
- "Truth Value" - when selected, a single True/False combo box will appears from which True or False can be selected
- "DateTime" - when selected, a single DateTime picker appears from which a DateTime value can be chosen.
- Possibly other WebGUI::Form items (if needed)
- "User Profile Field" - when selected, a new combo box appears containing the names of all wG User Profiling fields. The user is expected to choose one field from the list. The Operand will then evaluate to whatever value the User Profiling field equals for a given user
- "Flux Rule" - when selected, a new combo box appears containing the names of all Flux Rules. The user is expected to select one Rule from the list. The Operand will then evaluate to the Boolean status of the selected Rule for a given user (this allows Rules to be dependent on each other)
- "Group" - when selected, a new combo box appears containing the names of all User Groups. The user is expected to select one Group from the list. The Operand will then evaluate to True/False based on group membership (re-creates traditional Group-based authorisation)

I see many exciting possibilities in building Wobject-specific WebGUI::Flux::Operand modules, especially for the new breed of advanced Wobjects (Thingy, Survey 2.0 etc..). For example, a Wobject-bound Survey 2.0 Operand could check whether single Survey instances have been completed, while a general Survey 2.0 Operand could check whether *all* Surveys have been completed. Similarly, an advanced Wobject-bound Thingy Operand could allow smart checking against Thingy fields, opening up myriad possibilities.

At this point, unless any post-processors are going to be used, the first Operand is now completely defined.



WebGUI::Flux::Modifier

Modules in the **WebGUI::Flux::Modifier** namespace allow custom behavior to be registered against Operand return types. What this means is that if an Operand's return value matches a registered type, additional user input fields can be shown to the content manager before the Operand definition completes. This is important for flexible handling of DateTime objects (and probably other types that I haven't thought of yet).

Multiple WebGUI::Flux::Modifier modules can be registered against the same type. Here's how it works. When the content manager has finished choosing an Operand from the Operand combo box and filling in any additional user input fields, Flux checks whether any Modifier modules exist for the return type. If so, a new combo box is shown from which the content manager chooses which Modifier module they wish to use. Again, Modifier modules are encouraged to use appropriate names so that the Expression being built by the content manager reads naturally.

WebGUI::Flux::Modifier modules define:

- what name they should appear as
- what extra information (user input fields) the content manager should be prompted for when they are selected
- what dynamic help/tooltips should be displayed when they are selected

- what type of value they evaluate to

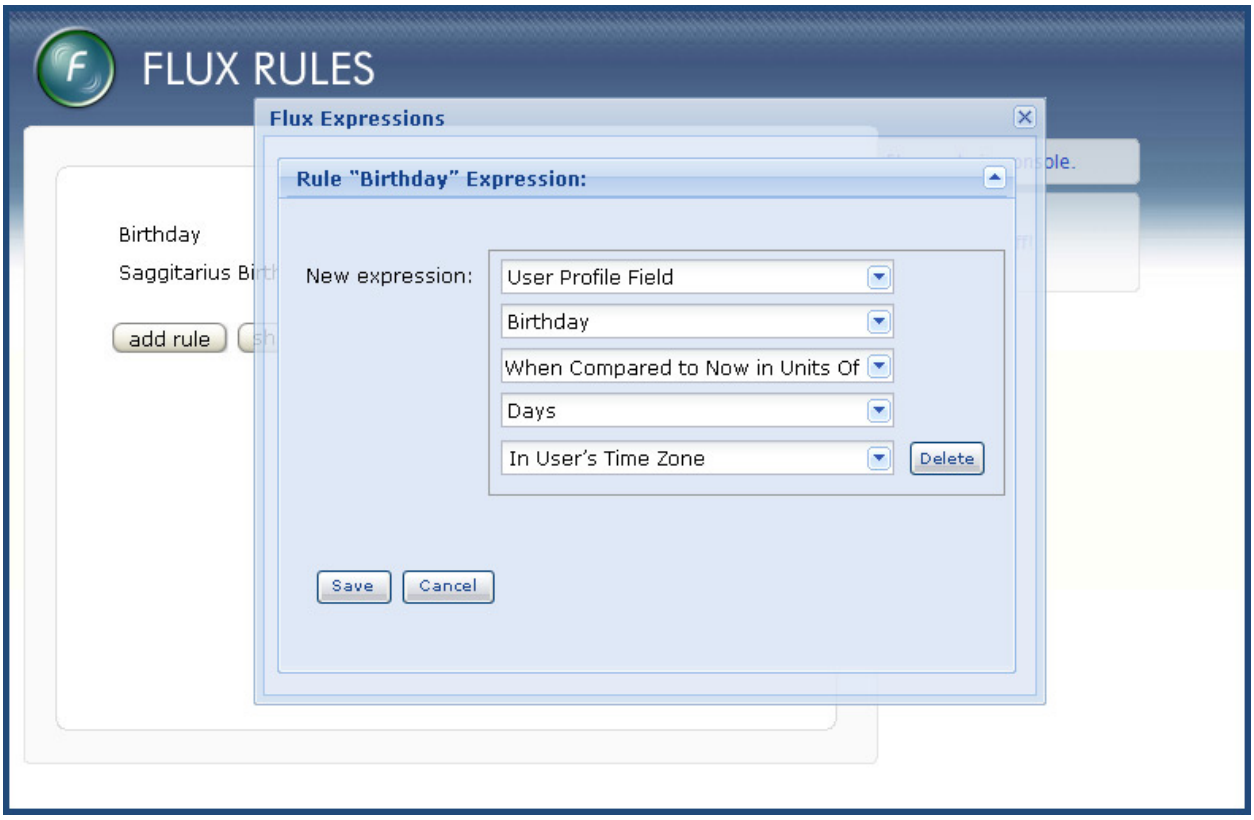
When evaluated, they have access to:

- The user for whom the Rule is being evaluated against
- The Rule being evaluated
- The values of any user input fields they prompted the content manager for

The following two WebGUI::Flux::Modifier modules will be registered against the *DateTime* type:

- "When Formatted As" - when selected two additional (mandatory) user input fields appear:
    - A text field into which the user can enter DateTime format patterns as per strftime() (e.g. "%Y-%m-%d" etc..) - when selected causes the DateTime to be formatted and returned as a string
    - A combo box containing all available time zones, including a special item called "In User's Time Zone" - when selected causes the Operand to be evaluated in the selected timezone
    - A helpful look-up table of available DateTime format patterns
- "When Compared To Now In Units Of" - when selected two additional (mandatory) user input fields appear:
    - A combo box containing all available DateTime units ("Years", "Months", "Days", "Hours", "Minutes", "Seconds", etc..) - when selected causes the DateTime to be subtracted from now(), and the resulting DateTime::Duration to be formatted as per DateTime::Duration::Format's format_duration() and returned as an integer
    - A combo box containing all available time zones, including a special item called "In User's Time Zone" - when selected causes the DateTime delta to be evaluated in the selected timezone

As can be seen below, post-processors allow more advanced manipulation of data types:



WebGUI::Flux::Rule::Operator

As soon as the first Operand has been defined, a new combo box appears from which the user can choose a Boolean infix **Operator**:

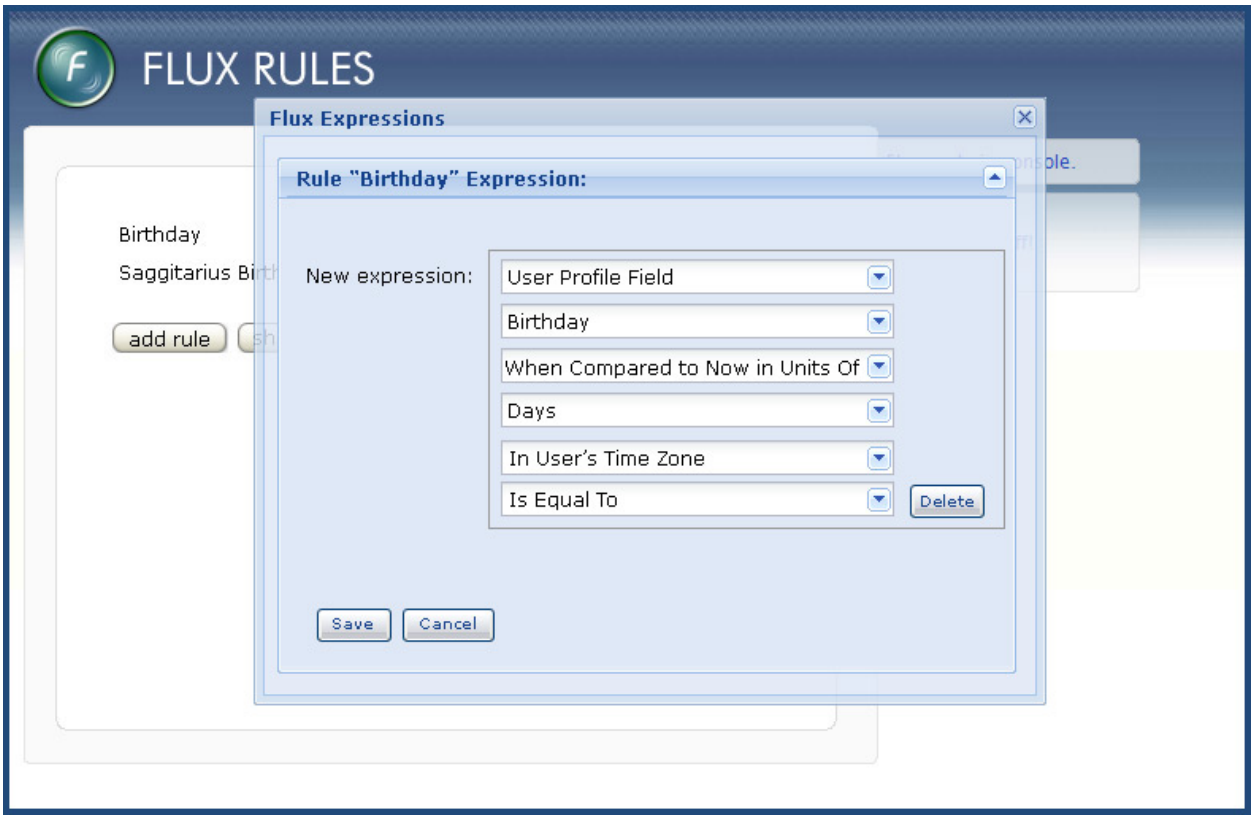
- Is Equal To
- Is Not Equal To
- Is Less Than
- Is Less Than Or Equal To
- Is Greater Than Or Equal To
- Is Greater Than
- Matches Partial Text
- Does Not Match Partial Text

Each of these Operators corresponds to a WebGUI::Flux::Rule::Operator module. These modules are quite simple, as they only need to define:

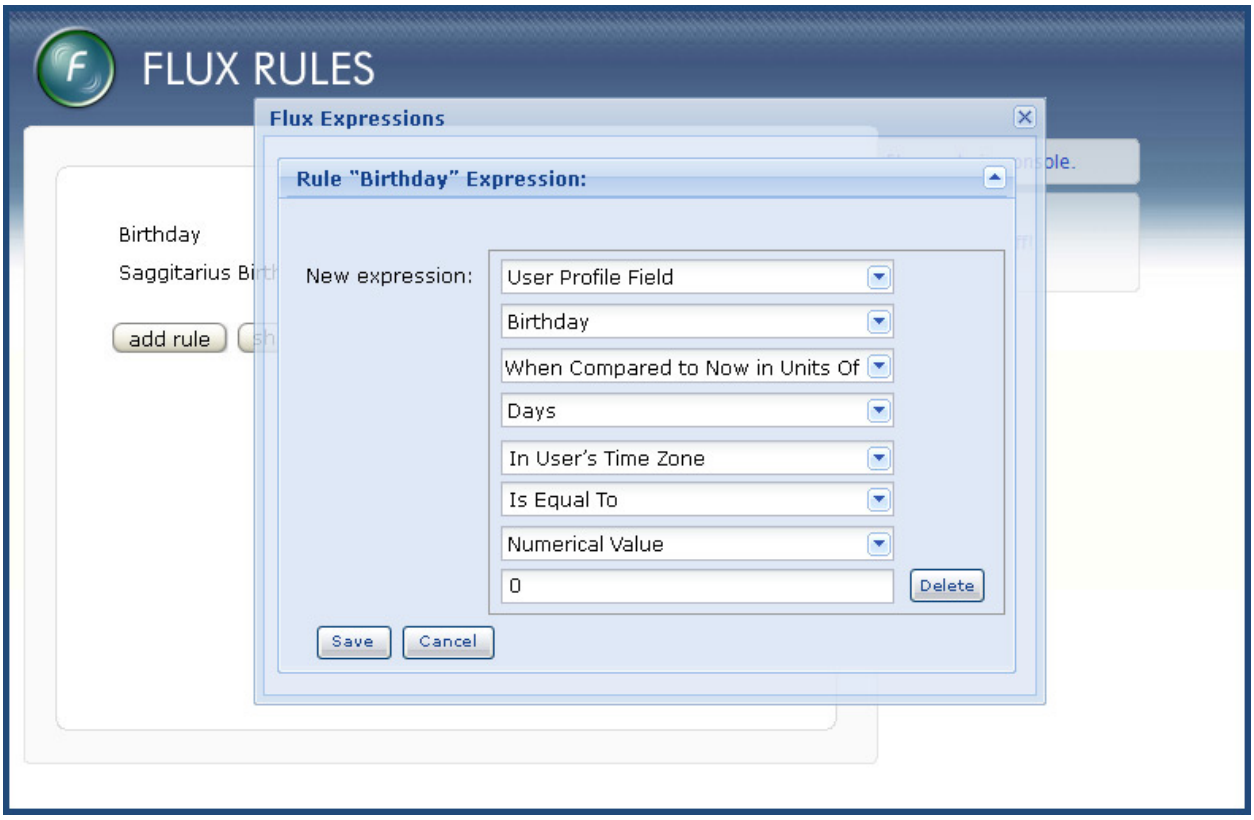
- what name they should appear as
- what dynamic help/tooltips should be displayed when they are selected
- a Boolean function that takes two arguments (the two Operands

More advanced Operators such as "Matches Regular Expression" can easily be added to Flux via new WebGUI::Flux::Rule::Operator modules.

An example of a partially-built Expression with an Operator chosen is shown below:



After the Operator has been chosen, the user is presented with a new Operand combo box from which to define the second Operand. This combo box behaves exactly as when defining the first Operand, except that it contains one extra special item called "From Suggested..". When selected, a new combo box will appear containing the list of suggested values defined by the first Operand. This will be useful to content managers when matching against something like a User Profile Field with a defined list of possible values, such as a Select Box.



The Expression has now been fully defined, and the Rule is usable. The flexibility of the system lies firstly in the fact that user fields, static values, other Rules and Wobject-specific properties can all be compared against each other in one consistent framework, and secondly in the way that the general mechanism for comparing Operands can be extended.

A table called **fluxExpression** will store data relating to each Expression. It will look like this:

- fluxExpressionId - primary key
- fluxRuleId - foreign key reference to the fluxRule table
- name :string - human readible name
- sequenceNumber - controls the viewing order, and also the 'e' number of the Expression
- operand1 - WebGUI::Flux::Operand::<module name> chosen by user for Operand 1
- operand1Args - JSON-encoded values of input fields requested by Operand 1
- operand1AssetId - if Operand 1 is bound to a Wobject, stores the relevant assetId
- operand1Modifier - WebGUI::Flux::Modifier::<module name> chosen by user for Operand 1
- operand1ModifierArgs - JSON-encoded values of input fields requested by Operand 1 post-processor
- operand2 - WebGUI::Flux::Operand::<module name> chosen by user for Operand 2
- operand2Args - JSON-encoded values of input fields requested by Operand 2

- operand2AssetId - if Operand 2 is bound to a Wobject, stores the relevant assetId
- operand2Modifier - WebGUI::Flux::Modifier::<module name> chosen by user for Operand 2
- operand2ModifierArgs - JSON-encoded values of input fields requested by Operand 2 post-processor
- operator - WebGUI::Flux::Operator::<module name> chosen by user

Multiple Expressions and the Combined Expression Editor

Multiple Expressions can be added to a Rule via repeated use of the "Add Expression" button and are listed one after the other as in the screenshot below. Expressions are numbered E1, E2, etc.. and can be re-ordered via drag and drop. In addition to their dynamic number, Expressions have an editable name that content managers can use for their own reference.

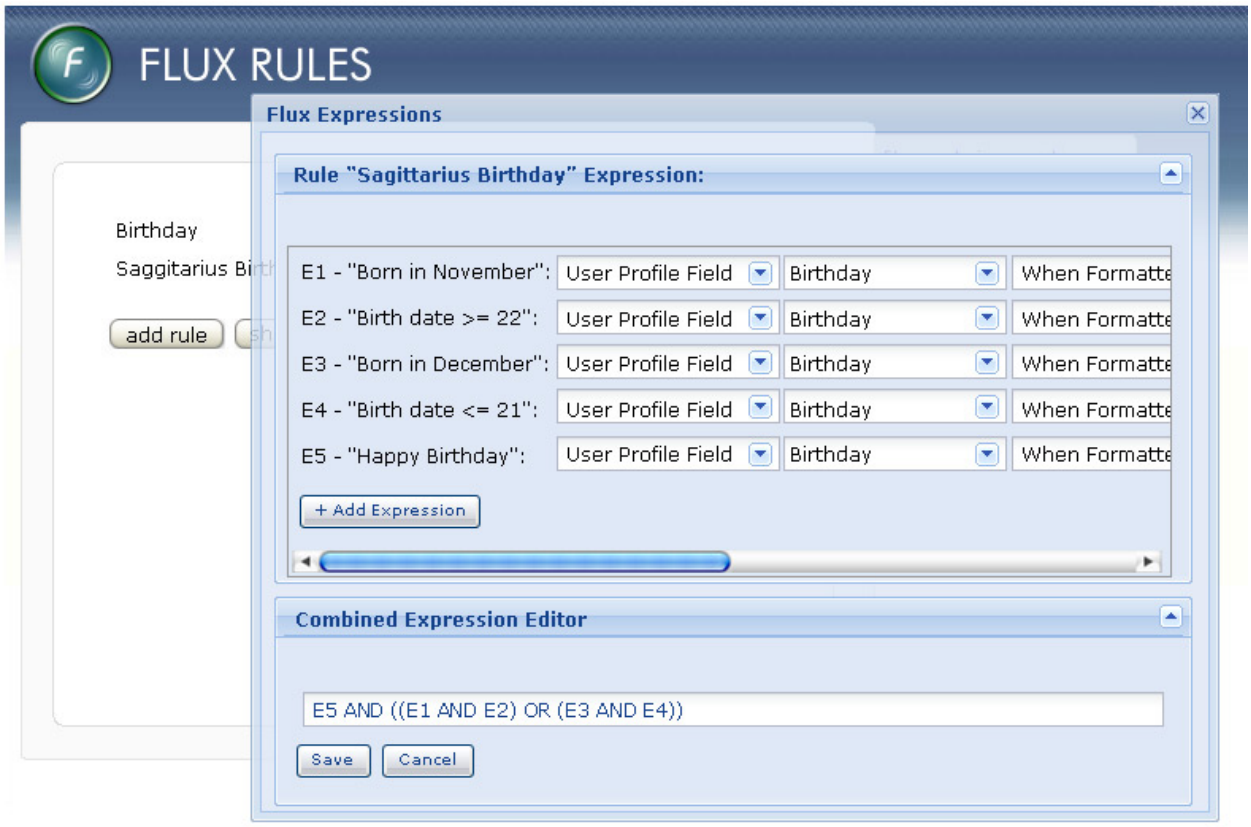
By default, multiple Expressions are AND'ed together to produce a single Boolean return value for the Rule. This is displayed dynamically in a text box at the bottom of the list of Expressions under the heading **Combined Expression Editor**. By default it will generate a Combined Expression that follows the general pattern:

E1 AND E2 AND E3

If content managers require more power, they can edit this text box directly.

- Permitted variables are E1, E2, .. corresponding to the number of Expressions defined
- Permitted operators are Boolean AND, OR and NOT
- Parenthesis are allowed

N.B. Ideally this would be a non-technical graphical interface however it seemed like too much work for the first version of Flux.

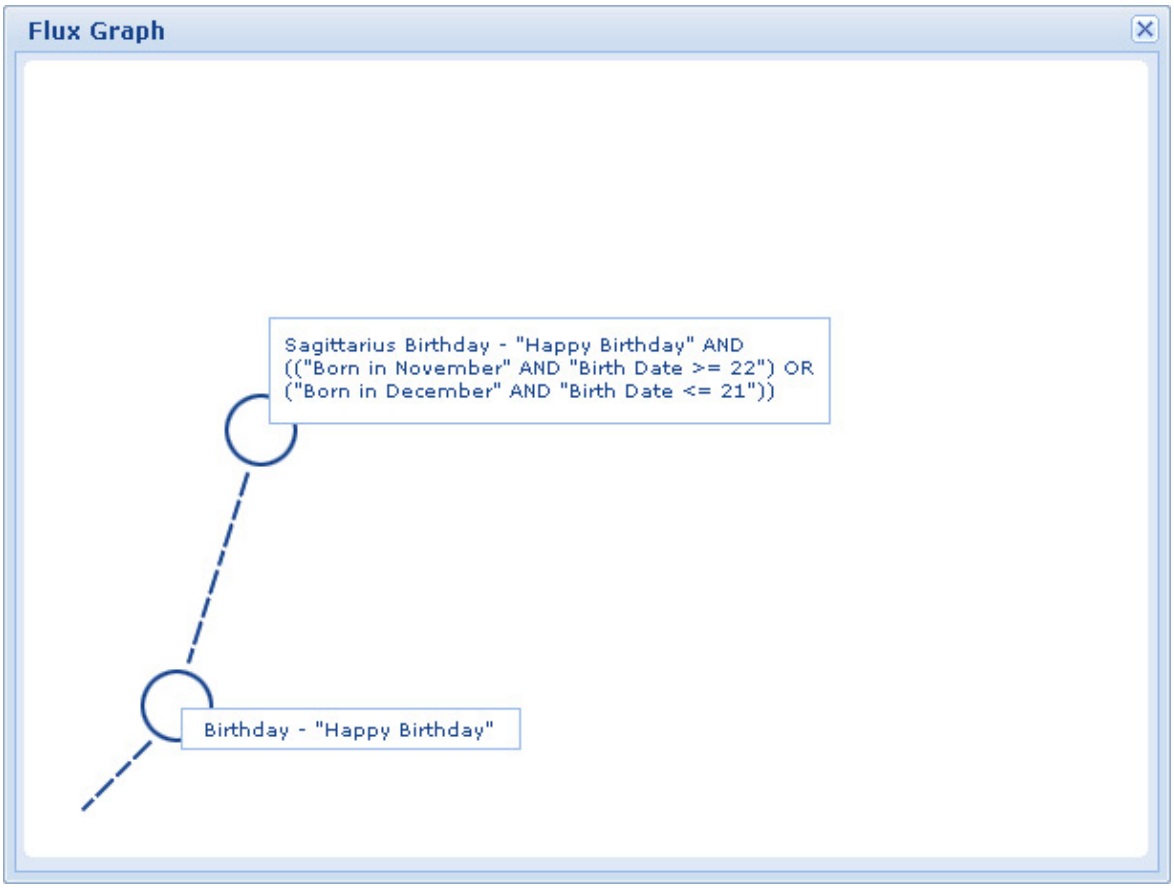


The Flux Graph

As mentioned above, Flux Rules can depend on each other, meaning that we end up with a **Flux Graph** of interconnected Rules. Obviously checking Flux Rules requires more CPU work than the normal WebGUI $user->isInGroup() API call, however there are many opportunities for optimisation. For example, the sticky flag can be used in many situations where users only ever gain access, rather then lose it. In situations where losing access is a possibility but rare, the sticky flag could still be used if custom code toggles user access manually.

N.B. The Flux Graph performs special checks to ensure that infinite loops don't occur.

It is easy to get the list of Rules and links into a standard format, which allows the Graph to be visualised using any standard Graph visualisation package (such as GraphViz). This Graph will be made available at any time from the "Show Flux Graph" button in the main Flux Rules Admin Console window. This feature is immensely useful to content managers when designing complex websites (and will be very impressive eye-candy). We annotate the Graph with as much information as possible, such as the name of each Expression contained in a Rule.



N.B. For visualisation purposes, it is not enough to just look for explicit references between Rules. Consider the situation where Rule A references Wobject-bound Operand X. Rule A should be visualised as being linked to whatever Rule(s) the Wobject instance is associated with. For example, consider a Survey Wobject instance with a Wobject-bound "Survey Completed" Operand. Rule A should be visualised as being linked (perhaps via a separate colour) to whatever Rules the "Who Can View" and other actions are set to on the Survey instance.