# Flow

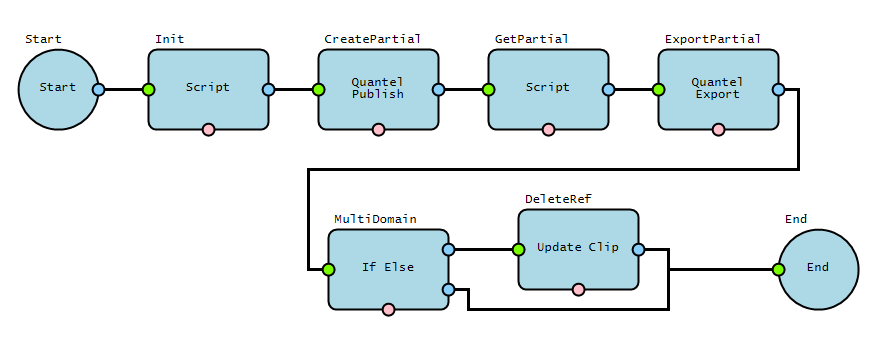
Flow is a simple workflow engine designed to be usable by people that understand their problem domain but don’t have extensive knowledge of complex workflow systems like BPMN2. Flow can be used to chain simple tasks together or can be used by more experienced users to construct more complex tasks with loops, branches and scriptable task blocks.

Flow is loosely based on the operation of the Amazon Simple Workflow Service and uses a lot of the same terminology. It is, however, completely self-contained and does not rely upon any external service. All data in flow is JSON and all inter-process communication is HTTP.

## The 10000 foot view

The first use of terms with a specific meaning is italicised.

A workflow is a JSON document that describes a collection of tasks, their connections, and the workflow variables. The workflow document also contains information that permits the workflow to be rendered as a diagram. Tasks represent a single item of work and are shown as symbols on the workflow diagram. Tasks can take inputs from the workflow variables and can return results to them. In this way, data can be transferred between different tasks as they execute. Normal tasks (i.e. not a start or end task) have connectors for an inflow, an error outflow and from one to three outflows. These connectors are joined by lines on the diagram, outflow to inflow, to define the order in which the tasks are executed. The completion of each task triggering the execution of another connected task until it triggers an end task.



The tasks can perform useful functions by initiating activities on workers. Each worker is a separate program that communicates using HTTP and periodically polls its tasklist to see if there are any activities for it to perform. Workers are usually associated with some external interface or device and can often perform a variety of appropriate activities. For example, an email worker might implement activities to send an email to an individual or group and to maintain a contact list.

A workflow is started by sending Flow a workflow start message. This creates an execution and initialises the variables of the execution with any input data from the start message. The execution also contains a history list which is a complete account of everything that happens to the workflow execution from start to end.

The workflow execution logic is implemented in the decider. The decider scans the history of each execution looking for workflow start messages, responses from workers or other external signals that will affect the workflow execution. The decider will update the execution history and will often create new a task on a tasklist.

The whole process of workflow execution is a chain of events (all recorded in the history) where a workflow start message causes the first task to be placed on the correct tasklist. The task is collected and the activity is performed. Then the results are sent back to Flow and are recorded in the execution history. The decider sees that the history has changed and by inspecting the history determines what task must happen next. It updates the history with its decisions and, usually, places another task on a tasklist to start the cycle again.

Note that tasklists are not owned by a workflow or execution. They are just a queue of jobs that one or more workers will poll for work.

At any time there might be many concurrent executions of a variety of different workflows. Nevertheless the Decider’s job is fairly simple. It just looks at each execution history that requires attention and decides what that execution should do next. It uses the workflow document for the execution to decide which task should follow the one just completed. Each executions history list serves a dual purpose. It provides complete information on the current execution status for the decider. It also provides a detailed audit trail for users so that they can see how the workflow progressed and what went wrong in the event of an error.

## Workers.

The Asynch, Delay, GenericUpdater, Script and Wait workers are general purpose examples that do not rely on any particular external interface or device.

Script workers execute JavaScript and can read and update workflow variables select between multiple outflows. It is trivially easy to import dotnet classes and functions into the script environment whether they are custom classes for a specific task or taken directly from the dotnet library. This makes it very quick and easy to prototype complex worker functions which are fit to be used in a production environment.

The Delay worker just does nothing for a specified time.

An updater is a special worker that reports back to some overseer system (presumably the one issuing workflow start requests) on job progress, completion and failure. The GenericUpdater worker is a simple skeleton for this function.

It is intended that all workers that take significant time to perform an activity should optionally implement an asynchronous mode. The task should have an optional signal name input. If a name is provided then the worker should not wait for the activity to complete but should return immediately with no results. This allows the workflow execution to continue with other tasks. At some later point in the workflow there should be a Wait task with the same signal name. When the long running activity is complete the worker sends a signal with the specified name and the results of the activity.

If the signal has already been received when execution reaches the wait task then the results are stored and the wait task completes. Otherwise the execution waits for the signal to arrive and then proceeds as before.

In this way it is possible to run a number of independent tasks in parallel and to wait for them all to complete. It is also possible to start “fire and forget” tasks by omitting the Wait task. The signal and the activity results will still be recorded in the execution history even though the workflow ignored them and whether or not the workflow has finished executing.

The Delay, Asynch and GenericUpdater workers are intended to be used as the basis of your own custom workers. Delay is a basic synchronous worker. Asynch is a basic asynchronous worker and GenericUpdater is the basis for an updater that can report back to your own systems with results and progress data. In time, it is hoped that there will be a library of the more common functions.

## Execution Variables

Execution variables are strictly unnecessary because the execution history is a complete record. It is, however, inconvenient and inefficient for humans and computers to scan the history for a specific result of a specific task. For this reason, each execution creates a physical copy of each named workflow variable and stores its current value as JSON in it. Variables can be initialised from the workflow start message to make the job parameters readily available. Variables with names starting with an underscore cannot be modified after being initialised and are a convenient way to safely store these parameters. Each result from a completed task can be stored in a variable. For example, a task might return three numbers named x, y and z. The task might assign the values of x and y to execution variables named fred and bill and throw the value of z away.

In the same way the named inputs to a task can be assigned from variables. It is also possible to assign task inputs with literal values or to use JSONPath to select a specific part of the value of a variable.

The reading and writing of variable values is handled by the decider when it starts tasks and processes task results.

## Workflows

Workflows are identified by a name and a version.

A workflow defines:

* The tasks and their connections, inputs and outputs.
* The workflow variables and their initialisation from a workflow start message.
* Some defaults for task timeouts.
* A description of the workflow.

The Tasks and connections also contain the physical layout and graphical elements used by Flow Monitor to construct a diagram in the editor. This information is completely ignored by Flow.

## Activities

Activities describe the available functions of workers. Each activity has a name and a version. The activity also defines:

* The expected inputs and outputs, their names, types, descriptions and default values.
* The names of the outflows (if there are more than one.)
* The default tasklist.
* Default values for activity timeouts.
* The description of the activity.

The only thing that ties an activity to a worker that can implement it is the tasklist name. The workers poll a tasklist and a task names a tasklist to receive its activity and inputs. If these don’t correspond correctly then the worker will not recognise the activity name. It is usual to accept the default tasklist in the activity definition. It may be helpful to think of the tasklist as the worker name even though this is not strictly true. The workers tasklist name is configured in its “worker.exe.config” file

One valid reason to override the tasklist in the task is to test a new version of a worker in its own temporary tasklist so that real jobs don’t attempt to use an untested worker.

## Database

Flow Core and the Decider(s) store all persistent data in an SQL Server database. This data includes the Workflow and Activity definitions, all of the data for active executions and the tasklists.

The database schema is very simple. This is made possible by storing the bulk of the data as JSON strings. In particular, the many, varied and likely to be extended, history events are stored in a table with a primary key, event type and timestamp common to all history events. The event parameters are stored in a JSON field which is unpacked into a complete object of the correct type when the event is fetched and converted back to the uniform JSON format when stored

## Detailed Workflow execution and customisation

A workflow execution is started by POSTing a Workflow Start Message to the REST interface at /start.

This message contains the name and version of a workflow and the workflow start data. The message is assumed to originate from some higher level system (the overseer) that controls the work dispatched to Flow and requires to be updated with job progress and completion messages. As a result there will likely be some site-specific data in every job that can be referenced by tasks in the editor task palette and by workers. For this example, assume that the following data is provided to every workflow:

\_config : object Workflow config

\_job : object Job data

\_jobType : string Job source/type

\_jobId : string External identifier

So the input data for any workflow at this site will consist of an object with properties named as above. The \_config data is static configuration data specific to the workflow. \_job is the input data for this particular job and \_jobType and \_jobId are the identifiers for this job in the overseer. The underscores have no particular function here but they match the names of the corresponding workflow variables which are made readonly by the presence of a leading underscore.

Note: \_jobID is recognised by FlowCore and will be copied into Executions so that the user can easily link a specific execution to an external job reference. Therefore, whatever the form of your external job references, they should be named \_jobID in your workflow start messages so as to take advantage of this feature.

All workflows at this site will define corresponding variables and initialise them from the start data. This ensures that the entire input data is made available to the workflow and cannot be overwritten. This initialisation phase is the only opportunity to copy these values. The start workflow message is written to the execution history but this copy is not available to workflow tasks.

The workflow is started by creating an execution and adding the start workflow message as its first history item.

The Decider sees the new workflow start message and creates variables from the workflow definition. It initialises the four site defined variables from the start message and then locates the first real task by following the outflow of the start task. It creates any inputs required by this task by evaluating literal data, reading a variable value or applying a JSONPath expression to a variable value. If this task has multiple outflows then their names are added to a special task input variable named $Outflows.

It records the scheduling of the first task in the execution history and adds the task with its input data to the correct tasklist.

Sometime later a worker polls this task list and is given the job. The job is marked with the workers unique id and the task started event is recorded in the execution history.

The worker completes the task and responds. A task completed history item is created with any returned results attached.

The decider sees the new task completed history item and finds the original task definition in the workflow definition. It updates the values of any output variables named in the task definition with the returned results.

The next task is identified, either by following the single outflow named “Out” or by following the outflow named by the special task output variable called $Outflow.

The next task is then scheduled as above and the process continues until an end task is reached. The workflow completion event is recorded in the history and the workflow execution is stopped because the Decider takes no action when seeing that event.

When a worker is running a synchronous activity that takes a significant time it is expected that it will send regular heartbeat messages back to FlowCore with an estimate of the activities progress. These messages are not recorded in the execution history but they are sent as notifications (an activity that never responds) to the updater worker. These notifications also contain the values of the \_jobId and \_jobType variables and site specific “progress data” that identifies the location of the current task in the overall job progress. For example, the full heartbeat notification might indicate that we are 36% of the way through task 3 of 5.

The updater is a special worker that is created to communicate with the overseer. As well as these automatic progress updates it can be assigned tasks in the usual way just like any other worker. In this way the overseer is kept informed about job progress and any required results.

The responses to heartbeat messages contain a cancellation flag which indicates that the current activity should be cancelled as soon as possible and an activity cancelled response returned. This happens when the execution has received a workflow cancellation request over the REST interface.

A significant feature of the Decider and workflow design is that there is only ever one active task in any workflow. The asynchronous worker feature means that there can be multiple simultaneous activities in a workflow but the tasks in a workflow are strictly serialised. This is due to the rules for task connections. Multiple outflows may converge on an inflow but each outflow must connect to precisely one inflow (zero or one for an error outflow.) As a result the flow can never split. This means that there is no contention for variable reads and updates and so the value of variables are always precisely defined. This fact also makes it easy to avoid contention in the database and permits the use of more than one decider if this is desirable for performance. Asynchronous activities allow safe concurrency in most practical cases with no loss of performance over a fully parallel design.

## Conditional tasks

Tasks can have up to four outflows. Most tasks have only one outflow, named “Out” and this will be followed automatically upon completion of the task. Every activity receives a special input named $outflows which is an array of the task outflow names. If this array has more than one entry then the activity must select one of these names and return it in a result named $outflow. The decider will then follow that outflow after the task completes.

## Error handling

Each task has an error outflow which is followed in the event of the activity responding with an activity task failed event. These can be left unconnected (unlike a normal outflow) and the activity failure will escalate to a workflow failure. If it is connected then the error outflow will be followed and the error ignored.

If the workflow contains a clean-up start event then this will be entered when the main workflow completes, with or without an error. The clean-up start should begin a separate workflow graph with its own end task. If an error occurs during clean-up then the clean-up will terminate immediately. It is assumed that the main workflow will count completed tasks or set flags in a variable so that the clean-up workflow know what needs to be reverted.

## Recoverable errors

Flow is designed to recover gracefully from network problems and crashes. All operations have timeouts and will automatically retry if they do not complete. External operations in the decider(s) and workers are guarded with a unique id to ensure that the results returned correspond to the data sent and late results returned when an operation is being retried are ignored. The critical point of failure is the database server which is known to be robust and reliable.

## Using the script worker

The script worker will execute arbitrary JavaScript using the Google V8 compiler. The script task has a $script variable which should be assigned with the script to be executed. Any other task inputs created by the user will appear as initialised variable in the outer scope of the script. Script results should be assigned as properties of the result variable which is also pre-defined in the scripts outer scope. Scripts should be edited by pressing the ellipsis button next to the Literal JSON field in the task input editor. This will open the proper editor where the script can be edited easily without having to escape the special JSON characters. To use the results in the workflow just create a task output with the same name as the property created on the result variable. See the “Adder” workflow in the sample database for a simple example of using a script to add two numbers.

Script workers can access the special $Outflows input variable which is an array of outflow names and can call the function setOutflow with one of the outflow names to select that outflow and branch the workflow execution.

# Flow Monitor

Flow monitor is an application that can be used to edit workflows graphically and to inspect workflows, executions, and their history, and variables and the tasklists.

## Editor

Selecting Workflows in the toolbar will display the workflow viewer/designer. The left hand pane shows a list of workflows in the database. Selecting one will display it in the document area. You can select tasks or the document area itself to browse the properties of the selected task or the workflow in the right hand pane. Three buttons at the bottom of the left pane will edit the selected workflow, delete it or create a new workflow. The Edit WF and New WF buttons will enter edit mode.

In edit mode the left hand pane changes to a palette of available tasks. Tasks can be dragged and dropped from the palette into the document and can be dragged around the document to arrange them neatly. Task connection points are drawn as coloured circles. Green for an inflow, blue for an outflow and red for an error outflow. Connections are made by dragging a rubber band line from a connection point to one with the opposite flow direction. Upon releasing the connector it will be auto routed. Connectors can be selected and connector segments can be dragged to adjust the layout if desired.

The document area is indicated as a white rectangle and will automatically scale with the diagram extents. The document area is also the area considered for connector auto-routing.

Dragging the document area will pan the document if it is larger than the window containing it.

In edit mode, the property editor in the right hand pane becomes editable. Select a task or the document background to edit the properties of that task or of the workflow. Workflow variables, task inputs and task outputs cannot be edited directly in the property editor. To edit these collections select the collection in the property editor and then press the ellipsis button ‘…’ on the right to open the appropriate collection editor.

At the bottom of the task palette are two buttons to save and discard your edits. Workflows are saved to the database using the Name and Version settings in the ID section of the workflow properties. Please ensure that these are set correctly. You should at least change the workflow version number if you want to retain the previous version.

## Workflow variables editor

To create or edit workflow variables first ensure that you are in edit mode and then click the ellipsis button that appears when you select Workflow Variables in the workflow properties editor. The Edit Variables dialogue will open. If you have defined no variables then you only option will be to Add one. Variables must have names composed of alphanumerics and underscores. A leading underscore indicates a readonly variable that cannot be changed after initialisation.

The Variable dropdown list lets you select a variable and the Type dropdown lets you select a datatype for the variable. The description area should be used to enter helpful text for users of the workflow. Variable can be initialised from the input data in the workflow start message or with a JSON literal value. To initialise from the input select the From Input radiobutton and enter a valid JSON Path expression where $ indicates the input property of the workflow start message. For example, using the scheme outlined above you must enter $.\_jobId to select the \_jobID property of the message data. The Required box can be ticked if the JSON Path must locate a valid value. Alternatively a default JSON literal may be given. The JSON Path can be left blank to leave the variable undefined. Select the Literal radiobutton to enter a literal JSON initial value.

## Task input editor

This enables task inputs to be assigned values from workflow variable in much the same way as the Workflow Variables editor. Task inputs are of two types. There are inputs intrinsic to a particular task symbol that cannot be deleted and can only have their assigned value changed. There are also user-defined inputs which the user can edit freely. User defined inputs are only useful in the script activity where they become initialised variables in the scope of the users script.

To assign a task input from a variable just select the Variable radiobutton and then select the variable from the list. You may enter a JSON Path to select some part of the variables value or just leave this blank to get the entire value. The Required and Default controls work as for variables.

## Task Output Editor

Task outputs are much like task inputs but simpler as you can only assign the entire value of a task input to a workflow variable. You can create user-defined outputs and again, these only make sense for script tasks where you can return arbitrary results from a script. You do not need to use every task result. Just set the variable name to (none) and the result will be discarded. Note that every result will be recorded in the workflow history so you don’t need to store results purely for diagnostic purposes.

## REST interface

Flow communicates with workers, the overseer system and Flow Monitor via a REST interface. This is implemented in Nancy. Response objects are shown as e.g. {a, b, c} which indicates an object with three properties with the given names and appropriate values.

A summary of the REST API follows

**GET /activities**

Get name and version of all activities

**GET /activities/{name}**

Get name and version of all activities with a given name

**GET /activities/{name}/versions/{version}**

Get the JSON definition of an activity

**PUT /activities**

Update or create an activity from a JSON definition

**GET /executions**

For each execution matching the query parameters, return { executionId, jobId, started} ordered by the started date. All query parameters are optional.

Query parameters

jobid=<string> Get the execution with this external job reference

workflow=<string> Get executions of the named workflow

after=<datetime> Get executions started after this time

state=<string> Get executions with the named state

The datetime string will be parsed by the dotnet DateTime type.

**GET /executions/{executionId}**

Return the full execution object as JSON for the single execution identified by the GUID executionId.

**POST** **/executions/{executionId}/signal**

**Signal the execution** identified by the GUID executionId. The body of the post must be a JSON WorkflowSignalledEvent object.

**POST /executionstate/executionid**

Change the state of a live execution

The request body should be - {"command": cmd}

Where cmd is:

“run” Resume a paused execution

“pause” Pause a running execution.

“cancel” Cancel an execution and attempt to cancel running activities.

“stop” Stop an execution immediately without ceremony (last resort use only)

**GET /history**

Get the history events for an execution

Query parameters

executionid=<string> The executionid (GUID) to get history for - REQUIRED

fromid=<int> Start after this event id

for=<int> Get at most this many events

signal=<string> Search for a named asynchronous signal in the history

**POST /start**

Start an execution. The body of the request should be a WorkflowExecutionStartedEvent.

The response is {executionId}

**GET /tasklists**

Get an array of tasklist names.

**GET /tasks**

**Get the tasks from named tasklist**

Query parameter

list=<string> The name of the tasklist to return tasks for

**GET /tasks/{taskToken}**

Get the definition of a single task.

**GET /variables**

**Get the variables from an execution as a map.**

Query parameter

executionid=<string> The executionid to return variables for

**Get /workflowconfig**

Get the configuration data for a workflow. If a version specific config exists then return it else return the generic config for the workflow name,

Query parameters

workflowname=<string> The name of the workflow to return config for. REQUIRED

workflowversion=<string> The version of the workflow to return config for.

**GET /workflows**

Get name and version of all **workflows**

**GET /workflows /{name}**

Get name and version of all workflows with a given name

**GET /workflows /{name}/versions/{version}**

Get the JSON definition of a **workflow**

**PUT /workflows**

Update or create an activity from a JSON definition

**DELETE /workflows /{name}**

Delete all workflows with a given name

**DELETE /workflows /{name}/versions/{version}**

Delete a **workflow**

## JSON Object definitions

The definitions of all history events, messages and workflow objects may be found in FlowShared/Json directory of the Flow project. Although the information in these files is dense it is also clearly presented and readable so these files are the formal definition of the schema. Since they are compiled into Flow they have the advantage that they are always correct and up to date.