# Scripting Documentation

The scripts in AnthillPro are written in BeanShell, which is a JSR-approved Java scripting language. Other scripting languages are also supported for select activities. The Scripting section of the User Documentation provides a basic introduction to scripting and AnthillPro.

## Script Types

While there are a large variety of uses for scripting, scripts come in two basic forms. The first is an in-line script. These are extremely short one line script fragments used to look up a variable value. The other type is a full script. These scripts may be as long as needed, often involving complicated logic.

### Variable Resolution

Because different projects, agents, and environments may have properties that build scripts and other actions need to be aware of, many of the settings in AnthillPro accept either a normal value or a specially formatted string indicating a variable should be looked up. These strings always begin with a dollar sign and enclosed by curly braces. The first part denotes the type of variable to be looked up, the second part indicates the variable. This is laid out like: `${type/variable_name}`.

So to look up an environment variable which specifies where the correct version of gmake is installed on a particular agent, the path entered might be: `${env/GMAKE_BIN}`.

The list of available variables (properties) on a particular agent may be viewed on the Variables tab of the Agent configuration (**System** > **Agents** > select an agent > **Properties**).

### Short Scripts

Short scripts are usually comprised of a single call to one of the script helpers (see Script Helpers). The classic example of this is to lookup the most recent stamp (version) applied to a particular build life and pass that to the build script as the version. The build script can then use that to name the binaries generated with the version number or inject that value into the executable. To do that for an Ant build script, enter the following script fragment into the build script properties: `-Dversion=${bsh:StampLookup.getLatestStampValue()}`.

The format for these scripts also requires a leading dollar sign with the contents enclosed in curly braces. The type specified is an indication of the scripting language to use (currently only BeanShell is supported), and the delimiter is a colon. The result of the script will be substituted back in. So if building version 1.0.5 of a project, the parameter `-Dversion=1.0.5` will be passed to the build script.

### Longer Scripts

Longer scripts are used wherever there is a multi-line text field that requests a script from the user. There is no requirement to wrap the script in any extra markers. AnthillPro will assume the contents of the text field are a script of the selected language. If there is no language selector on the page then BeanShell is the language option.

These scripts are used in locations where something interesting may need to be done. For instance, when creating notifications you may want to look up some information about a build that is not exposed to the template in an easy way. A detailed script could look this information up, process it into useful state and inject it into the Velocity context. These scripts also allow you to override AnthillPro's default options for incrementing build numbers to create a custom algorithm.

Like short scripts, these scripts may take advantage of the Script Helpers.

# The AnthillPro API

The API is available under the server installation through the web UI. The majority of the API is available at the tools page, including a copy of the AnthillPro API documentation (JavaDoc). The objects in the API strongly reflect what is seen in the user interface.

Looking up general information about Build Lives, workflows, and requests is often best done by using the dashboard helpers available in `com.urbancode.anthill3.dashboard` and the `DashboardFactory` in particular.

Other lookup helpers are available in the `com.urbancode.anthill3.runtime.scripting.helpers` package. These are discussed in some length in the scripting overview documentation.

To tap into the behavior of the system (rather than the data in the system), look at the options in the `com.urbancode.anthill3.services` package. The event service is of particular note, as many actions in the system fire events that are grabbed and acted on by listeners.

## Script Assumptions

Because there are some basics that are used by a large number of scripts, particularly the short scripts, AnthillPro will automatically import the classes of three (3) packages.

**You do not need to import these packages in your own scripts:**

- `com.urbancode.anthill3.runtime.scripting.helpers`
- `com.urbancode.anthill3.runtime.scripting.properties`
- `com.urbancode.anthill3.runtime.scripting.session`

## Script Helpers

Script helpers are available to every script and never need to be imported. They are always accessed statically, so to access them, follow the approach detailed in the Short Scripts section. Use the name of the of helper and the method name.

## Debugging Scripts

Debugging scripts can be tricky. For lengthy scripts, it may make sense to write the script first as a proper Java class in an IDE that has imported AnthillPro libraries. This can help you spot simple syntax errors without rerunning a script repeatedly.

The error messages presented by a failed script in the UI and log, often present a line number in the script where the error has occurred. It’s worth observing that if the text box you are looking at has wrapped a line, it is easy to count that line twice instead of once.

# Using the Script Library

Use the script library to create, organize, and provide security around often-used scripts. The basic organizational and security tool is the Script Group. The Script Group is a collection of AnthillPro scripts within the Script Library, and are often used to ensure that different departments/teams cannot edit each other's scripts (while ensuring each department/team may edit the scripts assigned to the Script Group they participate in). Once a Script Group is created and permissions set, assign AnthillPro scripts, listed on the System page under the Scripting menu, to a group.

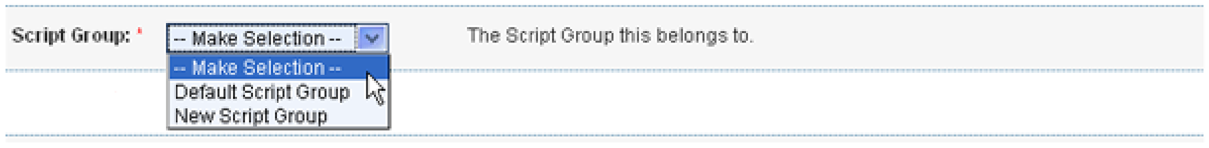
To use the Script Library, see Create a Script Group and Create a Script in the Script Library.

## Create a Script Group

The Script Group is a collection of AnthillPro scripts within the Script Library, and are used to ensure that different departments/teams cannot edit each other's scripts, while allowing each department/team to edit the scripts assigned to their Script Group. For example, large organizations often use Script Groups to create a boundary between development and QA teams, etc., based on the permissions assigned to the users in those environments. This allows developers and testers to modify the scripts they use, but does not allow developers to modify scripts used in QA and vice versa.

You must have administrative permissions to the System page in order to create a Script Group. To create a Script Group:

1. Go to **System** > **Script Groups** under the Script Library menu.
2. On the Script Group main page, click the **Create New** button.
3. **Name** the Script Group.
4. **Description.** Give an optional description of the scripts to be included in the group.
5. Click **Save**.
6. **Assign permissions.** To set which roles can access this Script Group, click the **View Security** icon under the Operations menu. Determine permissions and click **Save**. For example, a "tester" role may need to view an agent selection script, but should not be able to edit that script. In this case, assign the 'tester' role 'read' permissions but not 'write' permissions. Accordingly, a 'build master' role may need to use, edit, and set security for a script, so assign the build master all the permissions.
7. Click **Done**.
8. **Add Scripts to a Script Group.** Scripts in the Script Library (on the system page) are assigned to a Script Group. During script creation (see Create a Script in the Script Library), select the appropriate Script Group from the drop-down menu.

   To reassign a script to a new group, open the script configuration page and select the new Script Group from the drop-down menu.

## Create a Script in the Script Library

Script creation is performed on the System page, so make sure you have administrative permissions before continuing. AnthillPro ships with a set of default scripts that control how AnthillPro performs basic tasks; however, you will most likely want to create custom scripts to further define how AnthillPro behaves. To create a new script, go to the System page and select the appropriate script type from the Script Library menu. The process for creating scripts in the Script Library is the same for each script type; however, the actual scripts will vary.

To create a new Script in the Script Library:

1. Go to the **System** page and select the appropriate script type under the Script Library menu. Currently, you can create an new script for the following:
   - Agent Filter Scripts
   - Event Scripts
   - Job Pre-Condition Scripts
   - Post-Processing Scripts
   - Stamping Scripts
   - Step Pre-Condition Scripts
   - Workflow Priority Scripts
   - Working Directory Scripts
2. On the Script List page, click the **Create New** button.
3. Create Script:
   - **Name** the script.
   - **Description.** Give a description of the new script.
   - **Script Group.** Select the appropriate Script Group from the drop-down menu. See Create a Script Group.
   - **Language.** Select a scripting language from the drop-down menu.
   - **Script.** Input the body of the script. For instructions on writing the script, see either Agent Filter (Selection) Scripts, Event Scripts, Job Pre-Condition Scripts, Scripted Stamping, Step Pre-Condition Scripts, or Working Directory Scripts.
4. Click **Save**.
5. To organize and secure the script, see Create a Script Group.

## Agent Filter (Selection) Scripts

Scripted Agent Filters return any agent matching an arbitrary script. Usually this is used to check properties on the available agents (e.g., to find an agent with a specific tool installed or one that is used for a special purpose). The script-driven filter will also allow you to specify the number of agents to use. For simple jobs like builds, the default of one is appropriate; for load testing, where a set number would be useful, putting in a number like five would work better. You can also instruct AnthillPro to run the job on every matching machine in an environment. This is pretty typical for doing things like deploying static HTML to a load balanced set of front-end servers. The script knows how to select the front-end servers, and the number setting directs it to deploy to all of them.

Custom properties may be set by going to **System** > **Agents** under the Environment menu. Select an agent and create a new property on the **Properties** tab.

### Creating an Agent Filter Script

The script must evaluate to return a Where class or subclass, which is used to filter an array of agents to a subset of that array. Agent selection then proceeds from the array.

All the helpers (see Agent Helpers) implement the abstract `Criteria.filter()` method in a manner consistent with their name, and are directly available to the script. One may add the line `import com.urbancode.anthill3.runtime.scripting.helpers.*;` in order to access the normal scripting helpers and implement more advanced filters.

## Agent Helpers

AnthillPro Agent Helpers are classes used to help determine which agent to use.

### Where Helper

All agent selector scripts must return a **Where Helper**. Where Helpers can use other forms of criteria (listed further down).

- `Where.is(Criteria)` -- Allows agent that match the contained criteria.
- `Where.not(Criteria)`-- Allows agents that do NOT match the contained criteria.
- `Where.any()` -- Matches all agents.
- `Where.any(Criteria[])` -- Allows agents that match any of the criteria contained in the array.
- `Where.any(Criteria, Criteria)` -- Allows agents that match either of the criteria passed in.
- `Where.all(Criteria[])`-- Allows agents that match all of the criteria contained in the array.
- `Where.all(Criteria, Criteria)`-- Allows agents that match both of the criteria passed in.

### Variable Criteria Helper

The **Variable Criteria Helper** checks a variable on the agent and can be used by the Where criteria above.

- `Variable.isPresent(variable_name)`. Matches if the variable named is present on an agent.
- `Variable.equals(variable_name, value).` Matches if the variable named is on an agent and its value matches the value.
- `Variable.matches(variable_name, reg_ex)`. Matches if the variable named is present on an agent and its value matches the regular expression passed in.

### Advanced Criteria Helper

**Concrete criteria implementations are available and can be extended (as demonstrated in the advanced scripts above). The available choices:**

```
public class VariableEqualsCriteria extends Variable
    public VariableEqualsCriteria(String name, String value)
    protected VariableEqualsCriteria(String name)
    protected void setValue(String value)

public class VariableMatchesCriteria extends Variable
    public VariableMatchesCriteria(String name, String regex)
    protected VariableMatchesCriteria(String name)
    protected void setRegex(String regex)
```

```
public class VariablePresentCriteria extends Variable
    public VariablePresentCriteria(String name)

public class AndCriteria extends Where
    public AndCriteria(Criteria[] criteria)

public class IsCriteria extends Where
    public IsCriteria(Criteria criteria)

public class NotCriteria extends Where
    public NotCriteria(Criteria criteria)

public class OrCriteria extends Where
    public OrCriteria(Criteria[] criteria)
```

## Add JobConfig to Agent Filter Script

If the agent selection script is filtering for a workflow job, then the job configuration will be available in the context of the script as `jobConfig`. This allows access to the job in the agent filter script. See Agent Filter (Selection) Scripts.

## Example: Select Linux Server with Ant Installed Script

**This script will select a server that has Ant installed (by checking for the ANT_HOME environment variable).** The `Where.all()` requires all the criteria included to be matched.

```
return
  Where.all(
    Variable.isPresent("env/ANT_HOME"),
    Variable.equals("sys/os.name", "Linux")
  );
)
```

## Example: Manually Select a Platform to Build on Script

**This script expects the workflow to prompt the user for a ‘platform’ property and then matches that against the operating system type of the agents as supplied by the sys/os.name agent variable.** This is an advanced script that creates a new filter class and runs against it.

```
import com.urbancode.anthill3.domain.agent.Agent;
import com.urbancode.anthill3.runtime.scripting.helpers.*;

public class PropertyEqualsAgentVar extends VariableEqualsCriteria {
    String propertyname;
    public PropertyEqualsAgentVar(String propertyname, String varName) {
        super(varName);
        this.propertyname = propertyname;
    }

    public Agent[] filter(Agent[] agents){
        String platform = BuildRequestLookup.getCurrent()
           .getProperty(propertyname);
        this.setValue(platform);
        return super.filter(agents);
    }
}

return Where.is(new PropertyEqualsAgentVar("platform", "sys/os.name"));
```

## Example: Run on Same Agent as Originating Workflow Script

**This script is designed for non-originating workflows.** It runs on the same agent that the originating workflow ran on. The script ships as part of AnthillPro. To view the script, go to **System** > **Agent Filter** and select "From Previous Job" on the Agent Filter Scripts page.

```
import com.urbancode.anthill3.domain.agent.Agent;
import com.urbancode.anthill3.runtime.scripting.helpers.*;

Where myWhere = new Where() {
    public Agent[] filter(Agent[] agents) {
        // assumes that the build job was the first job in
             orig-workflow
        Agent targetAgent = BuildLifeLookup.getCurrent()
           .getOriginatingWorkflow()
           .getJobTraceArray()[0].getAgent();

        boolean found = false;
        for (int i = 0; i < agents.length; i++) {
            if (agents[i].equals(targetAgent)) {
               found = true;
               break;
            }
        }
        if (found) {
            return new Agent[]{targetAgent};
        }
        else {
            return new Agent[0];
        }
    }
};

return Where.is(myWhere);
```

## Example: Select Agents Based on Job-iteration Properties

Use the **Select Agents Based on Job-iteration Properties Script** to select the agent(s) for every job iteration based on the value of each job-iteration property. For example, a job with two iterations has a property with a name of "my_property" set on it. Each iteration property has a different value (i.e., iteration one has a value of "1" and iteration two has a value of "2"). In order for iteration one to run on a specific agent, you will need to set the same property-value pair on that agent (i.e., you will need to set "my_property" with a value of "1"). The same goes for iteration two. When the script is run, AnthillPro will evaluate the job-iteration properties and run each iteration on the agent with the corresponding property-value pair.

```
String propValue = PropertyLookup.get("my_property");
return Where.is(Variable.equals("my_property", propValue));
```

## Example: Return All Online Agents

Use the **All Online Agent Filter Script** to select every online agent when iterating a job. While this script returns all online agents, work will be delegated based on the number of times the job is iterated. For example, if the script returns 50 agents and the job has 7 iterations, only 7 agents will perform work.

To view the script, go to **System** > **Agent Filter** and select "All Online Agents" on the Agent Filter Scripts page. Below is a copy of the script that ships with AnthillPro:

```
import com.urbancode.anthill3.domain.agent.Agent;
import com.urbancode.anthill3.services.agent.AgentManager;
import com.urbancode.anthill3.runtime.scripting.*;
import com.urbancode.anthill3.runtime.scripting.helpers.*;
import com.urbancode.commons.util.CollectionUtil;

/* Return all agents online */
return new Where() {
    public Agent[] filter(Agent[] agents) {
        agentList = new ArrayList();
        for (agent : agents) {
            status = AgentManager.getInstance().getAgentStatus(agent);
            if (status != null && status.isOnline()) {
                agentList.add(agent);
            }
        }
        return agentList.toArray(new Agent[agentList.size()]);
    }
}
```

## Event Scripts

Event scripts are used to create an Event Filter that listens to (automatically thrown) events passing through the Event Service. An internal service manages listeners for events, and makes various AnthillPro activities available to the service. Including:

- `BuildLifeForBuildRequestStartedEvent` -- Announces a Build Life started based on a request.
  - `getBuildRequest()` -- Gives the BuildRequest that was fulfilled by a new build.
- `BuildRequestFailedEvent` -- Made available whenever a build request fails.
  - `getBuildRequest()` -- Gives the BuildRequest object for the failing request.
- `JobEndedEvent` -- Announces the end of a job.
  - `getJob()` -- Gives the job that ended.
- `JobStartedEvent` -- Announces the start of a job.
  - `getJob()` -- Gives the job that started.
- `StepTraceEndDateChangeEvent` -- Made available when a particular step ends.
  - `getStep()` -- Gives the step that ended.
- `StepTraceStartDateChangeEvent` -- Made available when a particular step starts.
  - `getStep()` -- Gives the Step that was started.
- `WorkflowEndEvent`
  - `getBuildRequest()` -- Returns the BuildRequest that caused this workflow to run.
  - `getWorkflowCase()` -- Returns the runtime information on the workflow that completed.
- `WorkflowStepEndedEvent`
  - `getBuildRequest()` -- Returns the BuildRequest that caused this workflow to run.

  - `getStepTrace()` -- Returns the runtime information on the executed step.
  - `getWorkflowCase()` -- Returns the runtime information on the workflow that ran this step.
- `WorkflowStepStartedEvent`
  - `getBuildRequest()` -- Returns the BuildRequest that caused this workflow to run.
  - `getStepTrace()` -- Returns the runtime information on the executing step.
  - `getWorkflowCase()` -- Returns the runtime information on the workflow running this step.

See also Using the Script Library.

## Event Script Extra Inputs

Once configured, the Event Script is immediately active and listening for events. The project, trigger, workflow and parameters provide references to the Event Trigger itself, as well as the workflow and project it belongs to.

**Event scripts are passed the following:**

| Name | Usage | Class |
|---|---|---|
| project | The configuration for the project in question. | `com.urbancode.anthill3.domain .project.Project` |
| trigger | Used to get a hook back to the using trigger. | `com.urbancode.anthill3.domain .trigger.event.Trigger` |
| workflow | The configuration for the workflow. | `com.urbancode.anthill3.domain .workflow.Workflow` |

## Event Script Returns

Event scripts are responsible returning an Event Trigger: `com.urbancode.anthill3.services.event.EventTrigger`

## Example Event Script

The script below, which triggers when any dependencies are completed, is a duplication of the one that ships with AnthillPro. It is provided here as a reference for commentary. The approach taken is to declare a new Java class, instantiate an instance of it, and return that to the Event Trigger that is trying to register itself. Matching events is a matter of using Field Criteria.

There are two default implementations of Field Criteria:

- **Field Equal Criteria.** Takes an event field and looks up the value of that field in the event. If it matches the criteria value provided, a match is made. Otherwise, it does not (see example below).
- **Field Member Of Criteria.** Creates a list of acceptable values and matches if the value in the event matches. For example, triggering if a status is failed or error rather that one or the other.

```
import java.util.*;
import com.urbancode.anthill3.domain.persistent.Handle;
import com.urbancode.anthill3.domain.trigger.event.*;
import com.urbancode.anthill3.domain.workflow.*;
import com.urbancode.anthill3.services.event.*;
import com.urbancode.anthill3.services.event.criteria.*;

    class MyEventFilter implements EventFilter {
        Handle triggerHandle = null;

        //------------------------------------------
        public MyEventFilter(EventTrigger trigger) {
            this.triggerHandle = new Handle(trigger);
        }

        //-----------------------------------------
        public Class getEventClass() {
            return WorkflowEndEvent.class;
        }

        //------------------------------------------
        public Criteria[] getCriteria() {
            Criteria[] result = null;

            EventTrigger trigger = (EventTrigger)triggerHandle.dereference();

            result = new Criteria[]{
                    new FieldMemberOfCriteria("workflow",
                            Arrays.asList(
                                    WorkflowLookup.getDependency
                                        WorkflowsFor(trigger
                                       .getWorkflow())
                            )
                    ),
                    new FieldEqualsCriteria("status",
                        WorkflowStatusEnum.COMPLETE)
            };

            return result;
        }

    };

return new MyEventFilter(trigger);
```

## Create a New Event Script

The Event Script is commonly used in conjunction with an Event Trigger that looks for actions such as a completed build of a related project or of an originating workflow with the auto-deploy flag set to "true." In order to create an Event Script, you must have Administrative permissions.

1. Go to **System > Event** under the Script Library menu.
2. On the **Event Scripts** page, click the **Create New** button.
3. Input script:
   - **Name** the script.
   - **Description.** Provide a description of the script.
   - **Language.** Select a scripting language from the drop-down menu. Currently, AnthillPro supports BeanShell,

Groovy, and JavaScript.

- **Script.** Input the script body. This script should return a Criteria object which will be used to evaluate the results of steps/commands. The object will determine if the command executed successfully or should be failed. Use of the Criteria used by static methods or classes.

4. Click **Save** then **Done**.

## Event Selectors

An Event Selector is responsible for inspecting events that pass through the event service for a workflow and returning true if a notification should fire. This could look at pretty much anything related to the event. For instance, if the selector is working on a workflow event and is intended for workflows that use tests supported by AnthillPro, it could examine the test results and notify if more than 10% of the tests failed, or if the tests took longer than expected.

AnthillPro ships with a number of Event Selectors that can be used as an example in creating custom selectors.

1. To view, go to **System > Event Selectors** under the Notification menu.

2. on the Event Selectors page, select the edit icon of the Selector of interest.

### Example Event Selector Script

**This is the default workflow success selector.** It finds that the event is related to a workflow, then checks to see that the workflow is both complete and successful.

```
import com.urbancode.anthill3.domain.workflow.*;

result = false;
if (event instanceof WorkflowEvent &&
    event.getCase().isComplete() &&
    event.getCase().getStatus().isSuccess()) {
  result = true;
}
return result;
```

### Usage Scenario: Event Selector (JUnit)

New event selectors can be created (if there are some more specific conditions that should be checked). One example would be to run a script if the build completed successfully but failures occurred in the JUnit tests.

**To add a new JUnit Event Selector, click the Create Event Selector button and provide the script below:**

```
import com.urbancode.anthill3.domain.workflow.*;

result = false;
if (event instanceof WorkflowEvent &&
    event.getCase().isComplete() &&
    JUnitReportHelper.getJUnitReportArray(event.getCase())!=null) {

  workflow = event.getCase();
  junitReports = JUnitReportHelper.getJUnitReportArray(workflow);
  for (int i = 0; i < junitReports.length; i++) {
     summary = junitReports[i].getTestSummary();
     if(summary.getPassingTests() != summary.getTests()) {
       result = true;
     }
   }
}
return result;
```

## Job Pre-Condition Scripts

Job Pre-Condition script are used to determine if a Job should run, and must return a Criteria object. Job Pre-Condition scripts may be created and/or edited at **System > Job Pre-Condition**. AnthillPro is packaged with a default library of Job Pre-Condition Scripts that will meet most needs; however, it is possible to add new scripts (see Creating Job Pre-Condition Scripts) based on other events. Currently, AnthillPro ships with the following Step Pre-Condition Scripts:

- **All Ancestor Jobs Success.** AnthillPro will run this Job only if all previous jobs succeed.
- **Always.** AnthillPro always executes this job, regardless of previous job success or failure.
- **Never.** AnthillPro will never run this job.
- **Parent Job Success.** AnthillPro will run this Job only if the parent job succeeds.

Note that when editing existing Job Pre-Condition Scripts, any changes made will be applied to *every job using that script*. It is usually best to copy the script of an existing Job Pre-Condition Script and paste it into a newly created script to avoid any deleterious effects on other projects.

### Job Pre-Condition Script Prerequisites

- You must have permissions to the System page.
- Understanding of AnthillPro scripting. See Scripting Basics.

### Creating Job Pre-Condition Scripts

To create and/or edit a Job Pre-Condition Script:

1. Go to **System > Job Pre-Condition** under the Script Library menu.
2. Click **Create New** on the Job Pre-Condition Scripts page to create a new script.

   If editing an existing Job Pre-Condition Script, select it from the Name menu or select the View icon under the Operations menu. When editing scripts, *all changes will effect every job that uses the script*.
3. Create/edit script:
   - **Name.** Give a name of this script. Once created, the name will appear in the Job Pre-Condition field during configuration.
   - **Description.** Give a description of this script. The description will appear on the Job Pre-Condition main page.
   - **Language.** Select a scripting language. Currently, AnthillPro supports BeanShell, Groovy and JavaScript.

- **Script.** Give the script, returning a Criteria object, that will determine if the job should run or not. See Custom Job Pre-Condition Scripts.

4. Click **Save**.

Once created, the Job Pre-Condition Script will appear on the Job Pre-Condition Scripts page. The **Used In** menu details what projects and/or library jobs use the script.

### Using Job Pre-Condition Scripts

Once a Job Precondition Script has been created, select it from the drop-down menu when adding a job to a work-flow.

### Custom Job Pre-Condition Scripts

It is also possible to create custom Job Pre-Condition Scripts based on other events. For example, the Job Pre-Condition Script below will only run the job if a particular status ("candidate") has been applied to the Build Life. If the Build Life does not have that status, the job will not run:

```
return new Criteria() {
 public boolean matches(Object obj)
    throws Exception {
 return BuildLifeLookup.getCurrent().hasStatus(StatusLookup.getStatusByName("candidate"));
 }
}
```

See also Status Selection Scripts.

## Post-Processing Scripts

A post processing script checks the exit code and log from an executed step and determines whether that step was a success or failure. A typical script might check for a non-zero exit code and declare failure. Or it might look for the string ‘Build Success’ and mark the step as a success.

The Post Process Script returns a Criteria object which is then used to evaluate the Command Result for each ant-hill-command in a step (if Criteria evaluates to false it will fail the command). All the helper methods return Criteria Objects.

See also Using the Script Library.

### Example Script

There are a handful of example scripts that ship with AnthillPro. **The script for Ant based systems checks for a non-zero exit code, as well as checking for ‘Build Failed’ for Windows machines that don’t return a proper error code.**

```
return Fail.unless(
    Logic.and(
        Logic.not(Output.contains("BUILD FAILED")),
        ExitCode.is(0)
    )
)
```

### The Helper Methods

The **Helper Methods** are fairly self explanatory. A list of them is below. These criteria will cause the build to fail if the criteria they contain is met.

- `Fail.always()`
- `Fail.never()`
- `Fail.on(Criteria a)`
- `Fail.onAny(Criteria a, Criteria b)`
- `Fail.onAny(Criteria[] a)`
- `Fail.onAll(Criteria a, Criteria b)`
- `Fail.onAll(Criteria[] a)`
- `Fail.unless(Criteria a)`
- `Fail.unlessAny(Criteria a, Criteria b)`
- `Fail.unlessAny(Criteria[] a)`
- `Fail.unlessAll(Criteria a, Criteria b)`
- `Fail.unlessAll(Criteria[] a)`

**Output Filters** inspect the output log from the command and return true if the patterns listed are found in the log.

- `Output.contains(String regex)`
- `Output.containsAny(String[] regexs)`

**Exit Code Filters** inspect the exit code from the commands issued and returns true if they match.

- `ExitCode.is(int exitCode)`
- `ExitCode.is(int[] exitCodes)`

To apply boolean operators to criteria, use the **Logic Helper**.

- `Logic.or(Criteria a, Criteria b)`
- `Logic.or(Criteria[] a)`
- `Logic.and(Criteria a, Criteria b)`
- `Logic.and(Criteria[] a)`
- `Logic.not(Criteria a)`

Some helpers take full regular expressions. The standard Java regular expression format is honored here. See Java

Documentation [http://java.sun.com/reference/docs/].

## Post Process Limitations

Post Process is not the solution to every need to determine success or failure because it is limited, run on the agents, and lacks access to helpers and files that we store on the server.

AnthillPro has a more powerful scripting step which can be used for added functionality. For instance, a testing suite will normally fail if any tests fail. If one wants to accept 5% of failures in a tool like JUnit, the post processing script will have to pass any number of failing tests. As a follow-up step, you could very easily use an "evaluate (BeanShell) script" step after the report publisher and use our JUnitReportHelper to get the object model of the JUnit tests and simply throw an exception (or print out something and use a post-process on that steps output to fail this step).

When success and failure can be determined by examining an exit code or some text in the log, the post-process script is excellent. When it requires knowing more about the project, other approaches are needed.

## Highlighting Lines of Interest in Log Files

You can use regular expressions to highlight lines of interest in log files. The only limitation is that in order to avoid loading multi-gigabyte output/log files into memory, AnthillPro does the matching on a line-by-line basis: therefore it is not possible to do an effective multiline pattern. For case sensitivity, AnthillPro follows Java convention (see http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html [???]). By default AnthillPro is case sensative; however, you can turn that (and other flags) on and off. For example, use: /(?i)error/ (@ee sections on "special constructs" and flags for doing things like matching whitespace, case sensitivity, unicode case-sensitivity, etc.).

Examples:

/foo.*bar|bar.*foo/

/^error (?!foo|bar)/ (matches 'error baz' but not 'error foo' or 'error bar')

# Stamping Scripts

A typical stamping strategy has a base number that is incremented either by the default algorithm in AnthillPro or a scripted updater. In contrast, a scripted stamping strategy uses some other logic to determine a stamp using the AnthillPro API.

See also Using the Script Library.

## Keeping a Build Number Script

**One approach for stamping promoted builds is to just rename a build from DEV 1.2.3 to QA 1.2.3 this script does exactly that:**

```
String currentStamp = StampLookup.getLatestStampValue();
int dev = currentStamp.indexOf("(DEV");

return "QA - " + currentStamp.substring(dev);
```

## Using a Changeset ID Script

**A relatively common approach is to take the most recent change-set or change-list number of the most recent commit a living build holds and use that as a stamp.** This has seen particular adoption in the Perforce and Subversion crowds.

```
import com.urbancode.vcsdriver3.*;
import com.urbancode.anthill3.runtime.scripting.helpers.*;

ChangeLog[] changeLogArray = ChangeLogHelper.getChangeLogArray();
int highestChangeset = 0;

// Look up the highest changeset id in the revision log.
for (int i = 0; i < changeLogArray.length; i++) {
 ChangeSet[] changesetArray = changeLogArray[i].getChangeSetArray();
 for (int j = 0; j < changesetArray.length; j++) {
   ChangeSet changeset = changesetArray[j];
   id = changeset.getId();
   // edit out the 'r' character for svn
   if (id.startsWith("r")) {
      id = id.substring(1);
   }
   int num = (new Integer(id.trim())).intValue();
   if (num > highestChangeset) {
     highestChangeset = num;
   }
 }
}

// If there is no changelog, look up the most recent failed and good build
// and take the highest number. We assume that the version number will
// be the first stamp.
if (highestChangeset == 0) {
 lastSuccess = BuildLifeLookup.mostRecentSuccess();
 lastFailure = BuildLifeLookup.mostRecentFailure();

 if (lastSuccess != null) {
   successStampArray = lastSuccess.getStampArray();
   if (successStampArray.length > 0) {
     stampStr = successStampArray[0].getStampValue();
     highestChangeset = (new Integer(stampStr.trim())).intValue();
   }
 }
 if (lastFailure != null) {
   failureStampArray = lastFailure.getStampArray();
   if (failureStampArray.length > 0) {
     stampStr = failureStampArray[0].getStampValue();
     int num = (new Integer(stampStr.trim())).intValue();
     if (num > highestChangeset) {
       highestChangeset = num;
     }
   }
 }
}

stamp = "" + highestChangeset;
return stamp;
```

## Scripted Stamp Update

For stamp updaters that should take an existing number and modify it, the basic updated often works fine. For custom updating of version numbers through, a scripted updater can be useful. This script must return the string value of the stamp to apply.

**This updater increments the third number found rather than the last one. So it will update 1.2.3.4 to 1.2.4.4.**

```
// The String 'input' is the current value
String current = input;
int startIndex = 0;
int endIndex = 0;

startIndex = current.indexOf(".", startIndex);
startIndex = current.indexOf(".", startIndex + 1) + 1;
endIndex = current.indexOf(".", startIndex);

long lastDigits = Long.parseLong(current.substring(startIndex, endIndex));
lastDigits ++;
String result = current.substring(0,startIndex) + String.valueOf(lastDigits)
   + current.substring(endIndex,current.length());

return result;
```

## Stamp Style Script

Stamp steps let you choose the name of the stamping style to use by script. There is nothing special passed into these scripts. The last line must evaluate to the name to be used. While you could do a sophisticated script here, an easy short cut is to name your script styles after whatever they are used for, like deploying to a particular environment. That way, simply looking up the Environment provides the stamping style to use.

**This script just looks up the environment the promotion is happening on and selects the stamping style of the matching name.** If the environment names are long, using the getShortName() method instead can be appropriate.

```
EnvironmentLookup.getCurrent().getName();
```

# Step Pre-Condition Scripts

Step Pre-Condition script are used to determine if a step should run, and must return a Criteria object. Step Pre-Condition scripts may be created and/or edited at **System > Step Pre-Condition** (see Creating Step Pre-Condition Scripts), or when the **Show Additional Options** link is selected during step configuration (see On-the-fly Step Pre-Condition Scripting).

AnthillPro is packaged with a default library of Step Pre-Condition Scripts that will meet most needs; however, it is also possible to add new scripts (see Custom Step Pre-Condition Scripts) based on other events. Currently, AnthillPro ships with the following Step Pre-Condition Scripts:

- **All Prior Success.** AnthillPro will run this step only if all previous steps in the job succeed.
- **Always.** AnthillPro executes this step every time the job is run, regardless of previous step success or failure.
- **Any Prior Failure.** If any previous job step fails, AnthillPro will run this step.
- **Never.** AnthillPro will never run this step.
- **Prior Failure.** If the immediately preceding step failed, AnthillPro will run this step.
- **Prior Success.** AnthillPro will run this step only if the immediately preceding step succeeded.

Note that when editing existing Step Pre-Condition Scripts, any changes made will be applied to *every step using that script*. It is usually best to copy an existing Step Pre-Condition Script and paste it into a newly created script to avoid any deleterious effects on other projects.

See also Using the Script Library.

## Step Pre-Condition Script Prerequisites

- You must have permissions to the System page.
- Understanding of AnthillPro scripting. See Scripting Basics.

## Creating Step Pre-Condition Scripts

To create and/or edit a Step Pre-Condition Script:

1. Go to **System > Step Pre-Condition** under the Script Library menu.
2. Click **Create New** on the Step Pre-Condition Scripts page to create a new script.

   If editing an existing Step Pre-Condition Script, select it from the Name menu or select the View icon under the Operations menu. When editing scripts, *all changes will effect every project and/or job that use the script*.
3. Create/edit script:
   - **Name.** Give a name of this script. Once created, the name will appear in the Step Pre-Condition field during step configuration.
   - **Description.** Give a description of this script. The description will appear on the Step Pre-Condition main page, and be displayed during step configuration when a user chooses to edit this script. For example, the description could include a warning not to edit the script, etc.
   - **Language.** Select a scripting language. Currently, AnthillPro supports BeanShell, Groovy and JavaScript.
   - **Script.** Give the script, that returns a Criteria object, that will determine if the step should run or not. See Custom Step Pre-Condition Scripts.
4. Click **Save**.
5. Once created, the Step Pre-Condition Script will appear on the Step Pre-Condition Scripts page. The **Used In** menu details what projects and/or library jobs use the script.

Alternatively, Step Pre-Condition Scripts may be created during step configuration. See On-the-fly Step Pre-Condition Scripting.

## On-the-fly Step Pre-Condition Scripting

Step Pre-Condition Scripts may be created and/or edited during step configuration, in addition to creating/editing Step Pre-Condition Scripts on the System page (see Creating Step Pre-Condition Scripts). To create a script during step configuration:

1. Select the **Show Additional Options** link on the step page.
2. If creating a new script, select the **New Script** link on the Pre-Condition Script field and proceed to Item Four.
3. If editing a script, select the existing script from the drop-down menu and then select the Edit Script link.

   Note that any changes made to an existing script will effect all steps that use the script.
4. Follow the procedures outlined in Creating Step Pre-Condition Scripts, Items Three and Four.
5. Once created, the Step Pre-Condition Script will appear on the Step Pre-Condition Scripts page. The **Used In** menu details what projects and/or library jobs use the script.

### Custom Step Pre-Condition Scripts

It is also possible to create custom Step Pre-Condition Scripts based on other events. For example, the Step Pre-Condition Script below determines if the step should run based on the value (`"true"`) of the property (`"MyProperty"`):

```
return new Criteria() {
    public boolean matches(Object obj)
  throws Exception {
      return "true".equals(PropertyLookup.get("MyProperty"));
  }
}
```

## Workflow Priority Scripts

Use workflow priority scripts to dynamically determine which workflow will run first. It is possible to set a dynamic workflow priority based on the Build Life (if applicable), environment, project, request, user, and/or workflow. The workflow priority script will run at the beginning of a workflow request, so if it fails for any reason the workflow request will fail, and the workflow will not run. See Creating a Workflow Priority Script and Using a Workflow Priority Script.

The priority, either High, Normal or Low, will be assigned to the workflow request based on the WorkflowPriorityEnum object given, and will be passed to all dependent workflow requests down the dependency graph. For example, if a workflow priority script sets a High priority based on the user making the request, and the requested workflow kicks off a build of a secondary workflow and a dependent workflow, those workflows will be assigned a High priority for the duration of the workflow.

### Creating a Workflow Priority Script

To create a workflow priority script:

1. Go to **System** > **Workflow Priority** under the Script Library menu. If you can't access the System page, contact your AnthillPro administrator.
2. Select **Create New** on the workflow priority list page.
3. **Name** the script. Give a name that reflects what this script does. For example, Environment Priority. The name will be used during workflow configuration/editing. See Using a Workflow Priority Script.
4. **Description.** Enter a description of this script.
5. **Language.** Select a language. Currently, BeanShell, Groovy, and JavaScript are supported.
6. **Script.** Enter the script that returns a WorkflowPriorityEnum object. Priority may be based on the Build Life (if applicable), environment, project, request, user, and/or workflow.
7. Click **Save** then **Done**.
8. See Using a Workflow Priority Script.

### Using a Workflow Priority Script

If the workflow priority must be determined dynamically, first create a custom workflow priority script (see Creating a Workflow Priority Script) before editing/configuring a workflow.

During workflow configuration, select the priority from the drop-down menu. Additionally, to change the priority of an existing workflow go to **Administration**, select the workflow, and click the **Edit Workflow** icon on the Main page. Select the new priority from the drop-down and click **Save**. The next time the workflow runs, the new priority will be used.

## Working Directory Scripts

Working directory scripts are created on the **System** page and are used by a project's source configuration to determine the directory the checked-out source will be placed in. Also, the set working directory step for a non-originating workflow job uses working directory scripts.

A working directory script is designed as a short script, so hard coded values may be entered for things like using a file system repository, or always running an installer from a particular directory. Nothing special is passed into working directory scripts, and the value generated is computed at workflow execution time on the agent in order to generate the working directory.

AnthillPro is packaged with a **Default Working Directory Script** that is available for every project that uses a working directory script (go to **System > Work Dir** under the Script Library menu to view). It, as well as the Anthill-Example working directory script, may be used as an example when creating new scripts.

When editing a working directory script, remember that *any changes you make to a script will effect every project using that script*. The easiest way to modify a script is to copy it and then create a new one with the changes.

- Not every SCM type uses a working directory script: some determine checkout by configuration, client specifications, etc. If your SCM type does not use a working directory script, that option will not be when you create a workflow.

Including any `${property:}` variables and `${bsh:...}` scriptlets in the working directory script gives you more control over how builds and deployments are carried out (such as setting up a separate directory for each Build Life, etc.).

See also Using the Script Library.

### Create and Use Working Directory Scripts

If a Set Working Directory job step is used, it will always override the Working Directory Script set on the originating workflow. For example, adding a job step that sets the Working Directory to `C:\Project_A\Subproject_01` will override the Working Directory (of `C:\Project_A\`) selected during Source Configuration. The job will always be run in the `C:\Project_A\Subproject_01` directory.

Ensure that the Working Directory is not set to something like `C:` or `C:\` because **the entire contents of the C drive will be permanently removed**. Also note that if the Working Directory of `C:\Project_A\` is set to clean up the workspace, the contents of `C:\Project_A\Subproject_01` will be removed as well. This may have an adverse effect on jobs that use the `C:\Project_A\Subproject_01` directory.

Before you start, make sure you have Administrative permissions to the System page.

1. Go to **System > Work Dir** under the Script Library menu.
2. Click the **Create New** button.
3. Create Script:
   - **Name** the script.

- **Description.** Give an optional description.
- **Script.** Input the path of the working directory in the text field. This may include any ${property:} variables and `${bsh:...}` scriptlets.

  *It is advisable to create a separate Working Directory Script for each project.*

4. Click **Save**.

### New Workspace for Each Living Build Script

**The following script is an example of how to set up a separate working directory for each build life.** It's notable in that it uses a mix of hard-coded and scripted paths, as well as using the path helper (makes paths safe for the agent operating system) and the build life lookup helper:

```
${anthill3/work.dir}/buildlife/
   ${bsh:PathHelper.makeSafe(BuildLifeLookup.getCurrent().getId())}
```

### One Workspace Per Project Script

**The standard working directory script also uses several helpers, but just creates a working directory for each project.**

```
${anthill3/work.dir}/projects/
${bsh:PathHelper.makeSafe(ProjectLookup.getCurrent().getName())}
```

## Build Life Note Scripts

A short BeanShell script is used to add a workflow property (requiring a user response) as a Build Life Note for deployments.

The script should look up the current Build Life and workflow, the user requesting the deployment, the environment, and the property that is being used to create a Build Life Note.

Additionally, the script must also include a string that adds the user's response as a Note on the Build Life. The example below uses a property called "Reason":

```
bl = BuildLifeLookup.getCurrent();
wf = WorkflowLookup.getCurrentCase();
user = wf.getRequest().getRequesterName();
env = EnvironmentLookup.getCurrent().getName();
reason = PropertyLookup.get("Reason");

note = user + " deployed to " + env + " to " + reason;
bl.addNote(note);
```

## Logging from a Script

For scripted logging, either create a log4j logger or write to the command output:

- **Use a log4j logger.** May be used whenever a script is evaluated by AnthillPro. See Logging from a Script (log4j).

- **Write to the command output.** Use in an Evaluate Script step. See Logging from a Script (Command Output).

See also **Tools** > **Scripting API > LogHelper**.

## Logging from a Script (log4j)

May be used whenever a script is evaluated by AnthillPro by creating a log4j logger. The output will typically be available on the Dashboard Job page when the Show Log icon is selected. However, under some circumstances (e.g., if used in a Post Processing script) output will be available on the agent's log, etc., depending on where the script is evaluated.

To log using log4j, use:

```
import org.apache.log4j.Logger;
Logger log = Logger.getLogger("script");
```

Then log at a level that is visible:

```
log.warn("This should appear in the log");
```

or

```
log.info("Only if info is enabled");
```

For information on creating a log4j logger, see Logger (Apache Log4j) [http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Logger.html].

See also **Tools** > **Scripting API > LogHelper**.

## Logging from a Script (Command Output)

Write to the command output with a **commandOutput** object. In the Evaluate Script Step script context, use `commandOutput.println("text");` to see the string in the command output.

See also **Tools** > **Scripting API > LogHelper**.

# Recipient Generator Scripts

A recipient generator is responsible for determining which users should be notified when the Event Selector determines a notification should go out. What is provided (supplied) to the script depends on whether the triggering event is related to a request, a workflow, or a task:

| Event Type | Supplied to Script |
|---|---|
| workflow event | workflow and request |
| build request event | request |
| task event | task, trace, workflow, and request |

**For notifications relating to a workflow, the following is put in the context:**

| Name | Purpose | Class |
|---|---|---|
| workflow | The execution of the workflow. | com.urbancode.anthill3.domain .workflow.WorkflowCase |
| request | The request that started the workflow. | com.urbancode.anthill3 .domain.buildrequest .BuildRequest |

**For notifications relating to a request, the following is put in the context:**

| Name | Purpose | Class |
|---|---|---|
| request | The request in question. | com.urbancode.anthill3.domain .buildrequest.BuildRequest |

Recipient generator scripts can also resolve project and workflow properties. This allows you to send notifications to an external list, etc., without having to set up a new scheme for each project. For example, if you set a project property (something like: project-mailing-list=myProjectTeam@myCompany.com) on each project, you can use that property to return your external e-mailing list in the recipient generator.

## Scripted Recipient Generator Examples

A fairly common usage pattern is to contact specific individuals, perhaps the system administrator, a manager, or a QA contact. To specify individual users, create a script like the one below.

**This example assumes that there is no LDAP integration in place and uses the default realm. If there is LDAP in place, use the appropriate realm.**

```
import com.urbancode.anthill3.domain.security.*;
userFactory = UserFactory.getInstance();
result = new User[] {
   userFactory.restoreForNameAndRealm("Eric","Anthill"),
   userFactory.restoreForNameAndRealm("dbeckham","Anthill")
};
```

- If the event selected is a `BuildRequestEvent` and not a `WorkflowEvent`, the script will not receive a workflow object. Instead, the script will be passed as a build request object named request. The same issue effects notification templates.

### Retrieves Two Specific Users

Make sure the last line of the script should evaluate to an array of User objects.

```
import com.urbancode.anthill3.domain.security.*;

userFactory = UserFactory.getInstance();
result = new User[] {
  userFactory.restoreForNameAndRealm("my_user","Anthill"),
  userFactory.restoreForNameAndRealm("dbeckham","Anthill")
};
```

### Retrieves the User Who Started a Workflow

Make sure the last line of the script should evaluate to an array of User objects.

```
import com.urbancode.anthill3.domain.buildrequest.RequestSourceEnum;
import com.urbancode.anthill3.domain.security.User;
if (RequestSourceEnum.MANUAL.equals(request.getRequestSource())) {
   return new User[] { request.getRequester() };
}
return null;
```

# Reporting Scripts

Reports are essentially an opportunity to use the API to gather data out of the system and display it in a custom way. A report is composed of no less than three scripts and a display template. The focus here will be on those three scripts. Those are:

- **Meta-data Script**
- **Report Script**
- **Template Context Script**

The Meta-data script is responsible for determining the inputs to a report as well as what data columns are displayed in that report. The report script is charged with looking up the appropriate data and collecting it as rows and columns for a report. The template context script is responsible for putting needed information into the context of the display template.

Our example report looks up the recent workflows in the last seven days for either all projects the user can see, or a specific project. It then displays a graphical breakdown showing which days were most active and which were more prone to be failure.

## DoW Example: Builds by Day of Week (Meta-data) Script

The meta data script is used to establish some of the guidelines of the report in question. This is a fairly advanced meta data script. Near the bottom, you see the most common component to a meta-data script which is the listing of the columns that will be used in the report.

The top section involves a parameter. Reports can demand a parameter which is always a simple String value. It can, however, be configured to prompt with options using a drop down box. In this case, we are asking which project should we run the builds by day of the week report on. So we have looked up all the projects in the system the user has access to, put them in a drop down list and added an all option to permit a view from a higher level.

```
import com.urbancode.anthill3.domain.project.*;
import com.urbancode.anthill3.domain.reporting.*;

ReportMetaData rmd = new ReportMetaData();

// Add some parameters to the report
SelectParamMetaData params = new SelectParamMetaData();

Project[] allMyProjectsArray = ProjectFactory.getInstance().restoreAllActive();
String[] labels = new String[allMyProjectsArray.length + 1];
String[] values = new String[allMyProjectsArray.length + 1];
for (int i = 0; i &lt; allMyProjectsArray.length; i++) {
   labels[i] = allMyProjectsArray[i].getName();
   values[i] = allMyProjectsArray[i].getId().toString();
}
```

```
labels[allMyProjectsArray.length] = "All";
values[allMyProjectsArray.length] = "all";

params.setLabels(labels);
params.setValues(values);
params.setName("project");
params.setLabel("Project");
params.setDescription("Select the project to report on.
   Or select 'All' for all projects.");

// Now we add our parameter options to the meta data
rmd.addParameter(params);

// Configure columns
rmd.addColumn("Monday");
rmd.addColumn("Tuesday");
rmd.addColumn("Wednesday");
rmd.addColumn("Thursday");
rmd.addColumn("Friday");
rmd.addColumn("Saturday");
rmd.addColumn("Sunday");

// Lastly, return the meta data
return rmd;
```

See also Adding Parameters to Reporting Meta-Data Scripts.

## DoW Example: Builds by Day of Week (Report Generation) Script

The report script is responsible for taking any parameters specified in the meta-data script and using them to look up rows of data that are compiled into a report. A report script is responsible for returning a ReportOutput object that contains all the data for the report.

The script below starts by setting a date range for looking up recent build lives, and uses one week. It then uses the property named project from the meta data script (which is populated into the script context automatically) to find the appropriate project to run against.

After that, it’s a simple matter of looking up the workflow summaries and counting the number of successful and failed workflows for each day in the week. Successful and failed builds are split out into separate rows which are then added to the report output.

```
import com.urbancode.anthill3.dashboard.*;
import com.urbancode.anthill3.domain.reporting.*;
import com.urbancode.anthill3.domain.userprofile.*;
import com.urbancode.anthill3.domain.workflow.WorkflowStatusEnum;
import java.util.*;

// Get the timezone for the current user
TimeZone timeZone = UserProfileFactory.getTimeZone();

Calendar cal = Calendar.getInstance(timeZone);
Calendar lastWeek = (Calendar) cal.clone();
lastWeek.add(Calendar.DATE, - 7);

// Figure out the project to use. "project" is the name of
// the parameter in the meta data script. It is provided here.
Long projectId = null;
if (project == null || project.equals("all")) {
   // leave as null
}
else {
   projectId = Long.parseLong(project);
}

// Get workflows for the last 7 days for the right project
DashboardFactory dashFact = DashboardFactory.getInstance();
BuildLifeWorkflowCaseSummary[] summaries;
summaries = dashFact.getBuildLifeWorkflowSummaries(projectId,
   lastWeek.getTime(), cal.getTime() );

// Initialize counts
int mondayCount = 0;
int tuesdayCount = 0;
int wednesdayCount = 0;
int thursdayCount = 0;
int fridayCount = 0;
int saturdayCount = 0;
int sundayCount = 0;
int failedMondayCount = 0;
int failedTuesdayCount = 0;
int failedWednesdayCount = 0;
int failedThursdayCount = 0;
int failedFridayCount = 0;
int failedSaturdayCount = 0;
int failedSundayCount = 0;

for (int i = 0; i < summaries.length; i++) {
   Calendar tempCal = (Calendar) cal.clone();
   tempCal.setTime(summaries[i].getEndDate());
   boolean failed = (summaries[i].getStatus() == WorkflowStatusEnum.FAILED
      || summaries[i].getStatus() == WorkflowStatusEnum.ERROR);

   int dow = tempCal.get(Calendar.DAY_OF_WEEK);
   switch (dow) {
      case Calendar.MONDAY:
          if (failed) failedMondayCount++; else mondayCount++;
          break;
      case Calendar.TUESDAY:
          if (failed) failedTuesdayCount++; else tuesdayCount++;
          break;
      case Calendar.WEDNESDAY:
          if (failed) failedWednesdayCount++; else wednesdayCount++;
          break;
      case Calendar.THURSDAY:
          if (failed) failedThursdayCount++; else thursdayCount++;
          break;
      case Calendar.FRIDAY:
          if (failed) failedFridayCount++; else fridayCount++;
          break;
      case Calendar.SATURDAY:
          if (failed) failedSaturdayCount++; else saturdayCount++;
          break;
      case Calendar.SUNDAY:
          if (failed) failedSundayCount++; else sundayCount++;
          break;
   }
}

// The output of a report is based on the meta-data
ReportOutput output = new ReportOutput(metaData);

ReportRow row = new ReportRow(output, "Failed");
row.setColumnValue("Monday", failedMondayCount + "");
row.setColumnValue("Tuesday", failedTuesdayCount + "");
row.setColumnValue("Wednesday", failedWednesdayCount + "");
row.setColumnValue("Thursday", failedThursdayCount + "");
row.setColumnValue("Friday", failedFridayCount + "");
row.setColumnValue("Saturday", failedSaturdayCount + "");
row.setColumnValue("Sunday", failedSundayCount + "");
output.addRow(row);

ReportRow row = new ReportRow(output, "Successful");
row.setColumnValue("Monday", mondayCount + "");
row.setColumnValue("Tuesday", tuesdayCount + "");
row.setColumnValue("Wednesday", wednesdayCount + "");
row.setColumnValue("Thursday", thursdayCount + "");
row.setColumnValue("Friday", fridayCount + "");
row.setColumnValue("Saturday", saturdayCount + "");
row.setColumnValue("Sunday", sundayCount + "");

output.addRow(row);

return output;
```

## DoW Example: Builds by Day of Week (Context)

The final script involved is the template context script. Once the ReportOutput has been created, a report template is responsible for presenting that data in an attractive or useful way. The template language used, is Velocity from the Apache Foundation [http://velocity.apache.org/]. Unfortunately, Velocity strongly limits the logic done within the template proper. To address this limitation, AnthillPro provides an extra BeanShell script that runs prior to the template execution. This script populates the Velocity context with anything needed, but does not have to return anything.

The report template script has access to some items that AnthillPro has already put in the velocity context. These include:

| Name | Purpose | Class |
|---|---|---|
| report | The report configuration. | `com.urbancode.domain .reporting.Report` |
| output | The data from the report | `com.urbancode.domain .reporting.ReportOutput` |
| request | The request object in case it is needed | `javax.servlet.http .HttpServletRequest` |
| response | The response object in case it is needed | `javax.servlet.http .HttpServletResponse` |

- **Graphical hooks.** There are hooks provided to display the data in graphical form. Most of the work in this case is done in the template script. The key here is to first define a report, and then request a URL for that report. The URL is then passed into the template which uses it to display the image. See `com.urbancode.anthill3.domain.reporting.graphing.GraphicsHelper` for details.

- **This script.** This script demonstrates how to set up a basic bar chart that displays the data in its various categories.

```
import com.urbancode.anthill3.domain.reporting.graphing.*;
import com.urbancode.anthill3.domain.singleton.serversettings.*;
import java.awt.Color;
import com.urbancode.anthill3.domain.reporting.graphing.*;
import com.urbancode.anthill3.domain.singleton.serversettings.*;
import java.awt.Color;

producer = new ReportDataProducer();
dataMap = new HashMap();
dataMap.put("data", context.get("output"));
report = context.get("report");
Color[] colorScheme = new Color[] {Color.RED, Color.GREEN}
chart = GraphicsHelper.createChart(producer, "stackedverticalbar3d",
      report.getName(),
   "Builds", report.getName(), dataMap, colorScheme);

String chartUrl = GraphicsHelper.getChartUrlString(context.get("request"),
   context.get("response"), chart, 600, 400);

context.put("chartUrl", chartUrl);
```

- **The template.** A simple report template that only contains the image designed by the context script is easy enough to produce. That URL where the chart is temporarily stored is provided to template for display to the users.

```
<html>

<head>
<title>$report.Name</title>
</head>


<body>
<center>
<h1> Report: $report.Name</h1>
<img src="$chartUrl"/>

</center>
</body>
</html>
```

## Adding Parameters to Reporting Meta-Data Scripts

To parameterize a report, attach either a SelectParamMetaData or TextParamMetaData object to the Report-MetaData object returned by the Meta-Data Script.

**SelectParamMetaData.** To display properties in a drop-down, add something similar to below:

```
Project[] allMyProjectsArray = ProjectFactory.getInstance().restoreAllActive();
String[] labels = new String[allMyProjectsArray.length + 1];
String[] values = new String[allMyProjectsArray.length + 1];
for (int i = 0; i < allMyProjectsArray.length; i++) {
  labels[i] = allMyProjectsArray[i].getName();
  values[i] = allMyProjectsArray[i].getId().toString();
}
labels[allMyProjectsArray.length] = "All";
values[allMyProjectsArray.length] = "all";

param.setLabels(labels);
param.setValues(values);
param.setName("project");
param.setLabel("Project");
param.setDescription("Select the project to evaluate successful and failed executions of t

rmd.addParameter(param);
```

**TextParamMetaData.** For simple text properties, add something similar to below:

```
TextParamMetaData param = new TextParamMetaData();
param.setName("enabled");
param.setLabel("Enabled");

rmd.addParamert(param);
```

## Scripted Lockable Resources

To dynamically select a Lockable Resource, use a BeanShell script that returns the name of a resource or a LockableResource object. Scripted Lockable Resources are configured during workflow creation, and return a resource exclusive to the environment you want the workflow to run in (e.g., a server name, etc.). When the workflow is run, it will then acquire the lockable resource, which may then be viewed on the Lockable Resources list. Scripted Lockable Resources are created with a maximum lock holder setting of one (1), so only a single workflow can acquire it at any given time.

For example, The following script creates/uses a Lockable Resource with the name of the environment the workflow is running in, plus the name of a server (i.e., "dev-server", "qa-server", "prod-server", etc.):

- `EnvironmentLookup.getCurrent().getName() + "-server";`

To script Resource Locks:

1. Go to **Administration** and select the **workflow** the Resource Lock is to be added to.
2. On the **Workflow Main** page, select **Add Resource**.
3. Set Lock:
   - **Explicit Resource.** Do not select a resource.
   - **Scripted Resource.** Give a BeanShell script that returns the name of a resource or a LockableResource object. If the name does not already exist, AnthillPro will create it once the workflow runs.
   - **Exclusive.** Check "Yes" to have AnthillPro exclusively lock the resource. When enabled, AnthillPro will obtain an exclusive lock for this resource, and override the maximum number of lock holders set during resource configuration.
4. Click **Save** then **Done**.
5. To add another Lockable Resource, repeat Items One thru Four.

## Scripting Notification Templates

AnthillPro Notification Templates are used to format information sent to team members regarding build status. To do so, the data is loaded into an Apache Velocity [http://velocity.apache.org/] template which is then processed and sent out.

While the Velocity template script has access to items AnthillPro puts in the Velocity context, Velocity's logic is limited. To extend the Velocity template logic, AnthillPro provides an extra BeanShell script (called **Context Script**) that runs prior to the Velocity template execution. See Example Notification Template Scripts.

- Before proceeding, if you are not familiar with Velocity templates, see the Velocity Documentation [http://velocity.apache.org/].

## Parameters

**For notifications relating to a specific job, the following is put in the context:**

| Name | Purpose | Class |
|---|---|---|
| trace | The trace of the steps executed in the job | `com.urbancode.anthill3.domain.jobtrace.JobTrace` |

**For notifications relating to a workflow, the following is put in the context:**

| Name | Purpose | Class |
|---|---|---|
| workflow | The execution of the workflow | `com.urbancode.anthill3.domain.workflow.WorkflowCase` |
| request | The request that started the workflow | `com.urbancode.anthill3.domain.buildrequest.BuildRequest` |

See **Tools > Scripting API** for additional variables that can be passed to the Velocity Template.

## Example Notification Template Scripts

Because the notification template for a job is necessarily different from that for a workflow, different template scripts are used. AnthillPro supports two basic Notification Template types: one for e-mails and one for instant messaging.

The same parameters are passed to both template types. When and how each type will be used is project and team specific. Depending on your needs, you will most likely have a combination of e-mail and IM templates notifying team members on a number of events.

- Any other AnthillPro actions that require either an e-mail or IM to be sent may fail unless the Notification Template is properly configured.

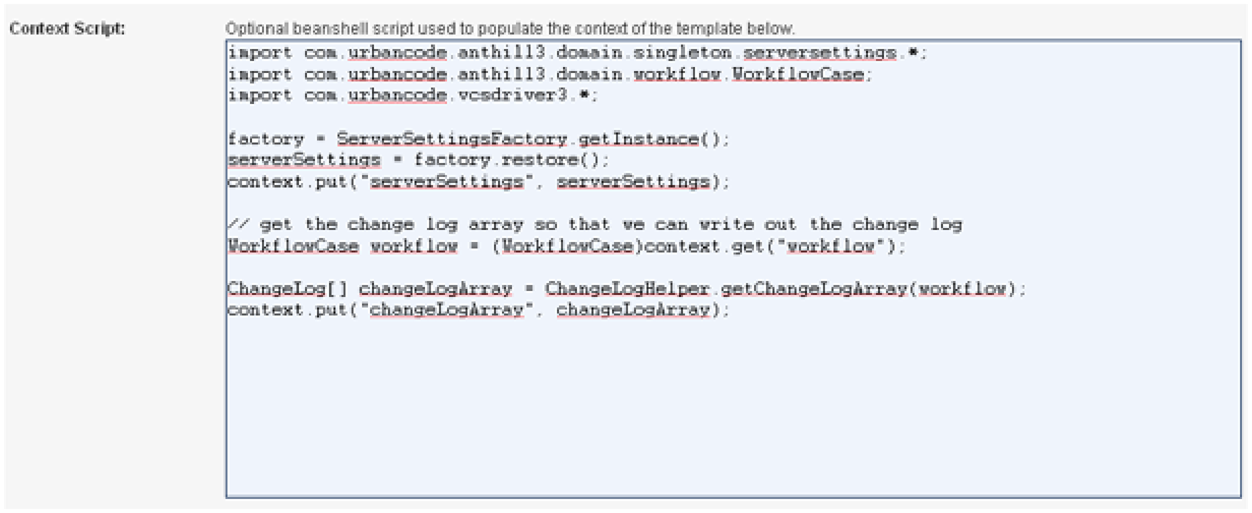
### Example E-mail Notification Template Script

The script for the **Simple Workflow Email Template** that ships with AnthillPro is pretty typical, and can be used as a prototype for creating other templates. It looks up the server settings so that the template can provide a link back to the project in question. It also looks up any change logs associated with the workflow so the changes in the build can be sent to interested parties.

1. To view, go to **System** > **Notification Templates** under the Notification menu.
2. On the Notification Template List page, select the edit icon of the **Simple Workflow Email Template**. (The Template may have a different name depending on what version of AnthillPro you are running.)

## E-mail Notification Template Context Script

The **Context Script** is a BeanShell script that AnthillPro runs prior to executing the Velocity template. The Context Script populates the Velocity context with anything needed, but does not have to return anything. The contents of the Context Script will vary depending on the type of information to be sent to team members.


Context Script:
Optional beanshell script used to populate the context of the template below.

```
import com.urbancode.anthill3.domain.singleton.serversettings.*;
import com.urbancode.anthill3.domain.workflow.WorkflowCase;
import com.urbancode.vcsdriver3.*;

factory = ServerSettingsFactory.getInstance();
serverSettings = factory.restore();
context.put("serverSettings", serverSettings);

// get the change log array so that we can write out the change log
WorkflowCase workflow = (WorkflowCase)context.get("workflow");

ChangeLog[] changeLogArray = ChangeLogHelper.getChangeLogArray(workflow);
context.put("changeLogArray", changeLogArray);
```

## E-mail Notification Template Text

The **Template Text** is where the Velocity Template is created. In addition to the standard Velocity text (see the Velocity Documentation [http://velocity.apache.org/]), the Template Text must import the AnthillPro-specific information from the BeanShell Context Script in order to deliver the appropriate information in the e-mail. This is accomplished at the beginning of the template body.

- The `## BEGIN SECTION Subject` and `## End SECTION Subject` lines are required for sending e-mail notifications, in addition to the `## BEGIN SECTION Body` and `## End SECTION Body` lines.

```
## BEGIN SECTION Subject

#set($project = $workflow.project)

Workflow: $project.Name - $workflow.Name
($workflow.BuildLife.LatestStampValue):
#if($workflow.Status.Name.equalsIgnoreCase('Complete'))
  Success
#else
  $workflow.Status.Name
#end

## END SECTION Subject

## BEGIN SECTION Body
## PROPERTY Content-Type: text/html
## PROPERTY X-Priority: 3
```

```
 ...
 ...

## END SECTION Body
```

Once the AnthillPro information is identified, it is incorporated in the Velocity Template. Complete Velocity Template creation is outside the scope of AnthillPro. For detailed instructions on Velocity Template usage, see the Velocity Documentation [http://velocity.apache.org/].

## Example IM Notification Script

Typically, IM templates are used to notify team members on workflow success or failure. The Template Text should be similar to what is below, and can be configured to return other events.

1. To view, go to **System** > **Notification Templates** under the Notification menu.
2. On the Notification Template List page, select the edit icon of the **Simple IM Template**. (The Template may have a different name depending on what version of AnthillPro you are running.)

- The `## BEGIN SECTION Body` and `## END SECTION Body` lines are necessary in order for the message to be sent. IM templates have no "Subject" lines, so make sure they are removed if you are modifying an e-mail template.

```
## BEGIN SECTION Body

#set($project = $workflow.project)

Workflow: $project.Name - $workflow.Name
( $workflow.BuildLife.LatestStampValue ):
#if($workflow.Status.Name.equalsIgnoreCase('Complete'))
  Success
#else
  $workflow.Status.Name
#end

## END SECTION Body
```

# Status Selection Scripts

The status scripts allow you to write a short script to select a status to either promote a living build to or get the change log since. The change log since script is designed so that when you promote something to an environment like QA you can hand the QA team a list of code changes since the previous deployment. That can give them clues about where to spend their time. This script currently uses the short script format. That means it must be a single line, return or evaluate to the name of the status in question and must use the `${bsh:}` format. The script can be several statements if necessary, but line breaks are forbidden.

Simply selects the status matching the name of the environment the deployment or promotion is occurring on. **For environments with long names, matching against the getShortName() method might make more sense.**

```
return EnvironmentLookup.getCurrent().getName();
```