

Podcast Producer: Writing Actions

Podcast Producer, an integrated part of Leopard Server, provides a complete publishing and management system for audio and video podcasts. Within Podcast Producer, workflows control the formatting and conversion of information from the source material into the final targets, whether it is a web page, a blog, or iTunes. Within these workflows, actions are the individual steps that perform the processing required by the workflow.

In order to write effective actions for Podcast Producer, you must understand the different properties that are executed from the command line. Individually, actions are discrete operations, but when chained together they provide a powerful processing environment so that the output from one action is compatible with the input of the next action within the process.

In this article, designed for Podcast Producer administrators developing their own workflows, you will learn about how to write and execute actions. Armed with this knowledge, you can easily develop your own actions to build different workflows. You will also learn about how to ensure that your actions are flexible and executable within the Xgrid environment and how to manage execution and resources during processing.

Workflow Actions and Properties

When developing and using workflows within Podcast Producer, the key elements of the workflow are the actions and the properties they use. The individual actions are part of the Workflow Definition File, and a typical workflow will include a number of different action steps that perform a sequence of operations on the material. For example, the workflow may be configured to first watermark the video, then generate different target formats (for iPhone, iPod, and AppleTV), and then finally publish the generated files. Each of these steps is an action. You can define sequences of individual actions by specifying dependencies for a given action. For example, you might specify that the dependency for publishing a podcast is that it has been processed into a suitable format first.

Actions use command line tools, both the standard set of applications built into Mac OS X and custom ones that you create to perform their operations. Writing an effective workflow action requires careful consideration due the way individual actions work with each other.

As mentioned in the first part of the Podcast Producer series, workflows define the process to convert incoming raw material into the final content to publish and create the environments, blog pages and other material that make your podcast available to a variety of different targets. The main configuration of the action list and operations is within the `template.plist` file, which can be an XML file or an old-style (non-XML) ASCII file. When processing a podcast through a workflow, the workflow uses information derived from a combination of different properties to process the podcast. These properties fall into three categories: default, custom and dynamic.

The default properties are configured through the Podcast Producer Administration Interface, and these properties set information to be shared by all workflows, such as copyright notices and common tools. The custom properties are also set through the Podcast Producer administration interface, setting properties unique to your installation. For example, you can use these properties to define the location of introduction movies used to provide introductory material. The dynamic properties are set automatically at the time a podcast is submitted for processing, and contain unique information about the job, such as the title and description of the podcast, and the user who submitted the information (see **Table 1: Main Dynamic Properties** for a list and description of the main dynamic properties).

Unprotected properties are available when executing actions by placing the name of the property within a double-dollar sign. For example, the `Organization` property is available within actions by quoting the property as `$$Organization$$`. On the other hand, encrypted properties are accessible using one-time passwords and the double-pound sign.

Executing Actions

When a podcast is submitted to Podcast Producer for processing through a workflow, the action called preflight within the Workflow Definition File is executed first. In the standard workflows supplied with Leopard Server, the action preflight executes the script defined by the `Preflight Script Path` global property.

The flow of actions and execution after that initial preflight action are determined by parsing the action specification and working out the dependency graph required to reach the postflight action, the final action within the workflow. The postflight action by default executes the script configured in the `Postflight Script Path` global property.

The dependency graph is determined automatically by Xgrid by building a list of the dependencies for each action and tracking back from the postflight action. For example, here we have the contents of a simple three-step Workflow Definition

Related Articles

Table 1: Main Dynamic Properties

Podcast Producer: Anatomy of a Workflow

Podcast Producer: Using the Command Line

Podcast Producer: Publishing on YouTube

Podcast Producer: Scheduling Podcasts

Leopard Technology Series for Developers: Server Overview

Resources

Reference Library: Mac OS X Server

Mac OS X Server Podcast Producer Workflow Tutorial (PDF, 1MB)

Mac OS X Server Podcast Producer Administration (PDF, 2.1 MB)

Leopard Server QuickTours (in iTunes)

Quick Tips Videos

Podcast Producer Mailing List

Podcast Producer Technology Brief (PDF, 1.2 MB)

File:

```
(
  {
    taskSpecifications = {
      preflight = {
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
      mail = {
        dependsOnTasks = (preflight);
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
      postflight = {
        dependsOnTasks = (mail);
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
    };
  }
)
```

From this definition, the postflight action depends on the mail action, and the mail action depends on the preflight action. Therefore, the execution sequence would run as follows: preflight, mail and postflight.

The process of determining the dependencies in an action sequence can be simplified by drawing a diagram of the sequence. This diagram can then be used to define the actual dependencies within the template.plist file. You can see an example of a simple workflow execution model in Figure 1.

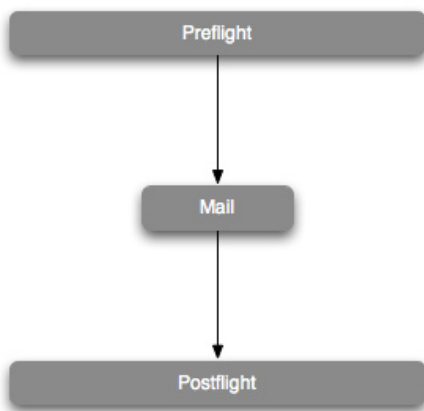


Figure 1: A simple Workflow Action Sequence showing dependencies

For more complex models, it becomes even easier to understand the sequence and dependencies. Figure 2 shows a more detailed diagram.

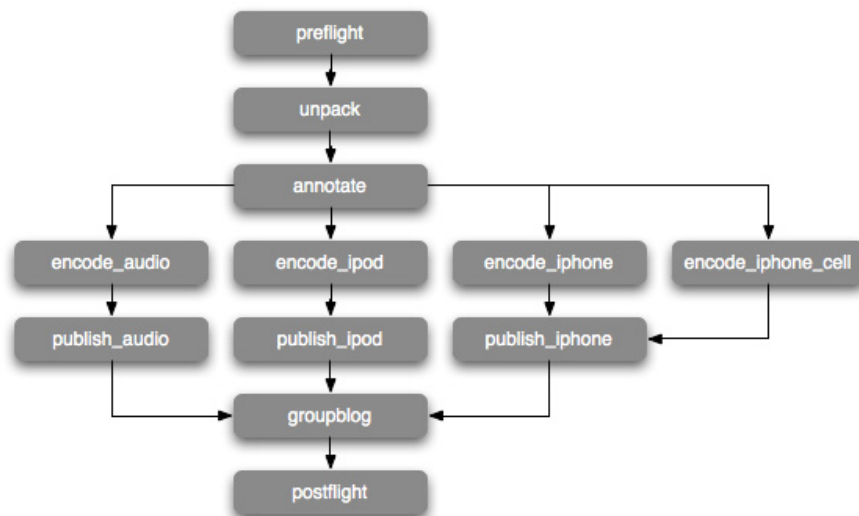


Figure 2: A complex Workflow Action Sequence showing dependencies

For this workflow, the podcast is being encoded suitable for use on a blog with different audio and video formats.

Building a Basic Action

Although actions have a specific sequence, it is the individual action item that performs the actual processing. When defining an action, the first consideration should be the task that needs to be completed, the required source information and the desired result. When processing the source podcast file into the target format, for example, the inputs are the source file and the target format, and the destination is the file where you want the converted audio or video to be created. For the destination file, the name of the file that you choose is important because it should be both identifiable as being part of the submitted workflow, and it should also be identifiable by the other actions within the workflow as a whole.

In most cases, you can use the Universally Unique ID (UUID) generated by Podcast Producer as a unique identifier. When converting podcasts into different formats, the convention is to use the source filename as the basis for the target filenames for each action. By combining the `$$Content File Basename$$` property with other text you can create as many different files as you want to help describe the inputs and outputs of your action.

For example, Figure 3 shows a typical workflow for publishing an iPod encoded podcast to the website.

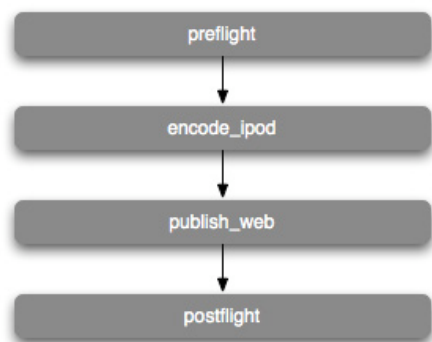


Figure 3: Workflow Action Sequence for publishing an iPod podcast to the Web

Using the graph shown in Figure 3, you can create an action sequence that first takes the source file and generates the target file that will be published to the website. Then, for the web publishing action, we take the generated file and use that as the source for the filename as published to the website.

To define an action, you specify the dependencies, the command to be executed, and the arguments to the command. The entire command (command name plus arguments) will then be executed. You can work backwards from the equivalent command line to help define the action contents. For example, when encoding a podcast into a new format, you use the `pcastaction` command with the `encode` task:

```
$ pcastaction encode --basedir=basedir --input=sourcefile --output=destfile --encoder=ipod
```

As an action you would define this in script form as follows:

```
encode_ipod = {
  dependsOnTasks = (preflight);

  command = "/usr/bin/pcastaction";
  arguments = (
    encode,
    "--basedir=basedir",
    "--input=sourcefile",
    "--output=destfile",
    "--encoder=ipod"
  );
};
```

Each individual argument is treated as a string within an array of arguments. When using an XML based definition, this is clearer, as you can see here in Figure 4. There is a specific arguments property (which is defined as an array), and each element of the array equates to an argument on the command line.

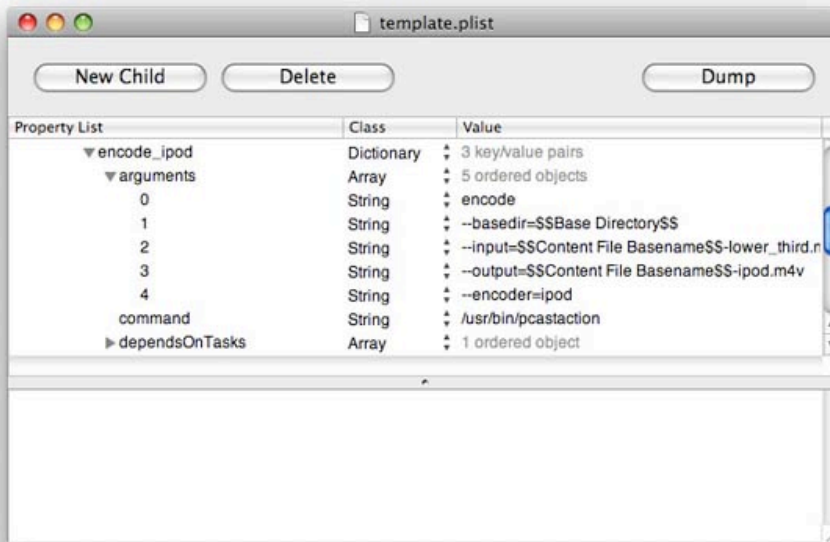


Figure 4: Defining action parameters within an XML property list

The final action, with the properties that we need inserted into the arguments, demonstrates how an action is called and executed:

```
encode_ipod = {
  dependsOnTasks = (preflight);

  command = "/usr/bin/pcastaction";
  arguments = (
    encode,
    "--basedir=Base Directory",
    "--input=Content File Name",
    "--output=Content File Basename_ipod.m4v",
    "--encoder=ipod"
  );
};
```

Note here how the `Content File Name` variable has been used. This is one of the standard variables automatically created by Podcast Producer when an action is executed. The `Content File Name` variable is a shorthand for the full name of the original content, and is therefore equivalent to using `Content File Basename.Content File Extension`.

Chaining Actions

When chaining actions together, it is up to the workflow creator to ensure that the correct values are used in each action for inputs and outputs. The web publishing action will need to use the value used in the `--output` command line argument as part of the Publishing Definition File. There is no automatic way of defining this; instead you must ensure that the contents

of your Action Definition File are described correctly. For example, the web publishing action should contain the right specification:

```
publish_ipod = {
  dependsOnTasks = (encode_ipod);

  command = "/usr/bin/pcastaction";
  arguments = (
    publish,
    "--basedir=${Base Directory}",
    "--web_root=${Web Document Root}",
    "--web_url=${Web URL}",
    "--date=${Date_YYYY-MM-DD}",
    "--title=${Title}",
    "--format=ipod",
    "--type=video/mp4",
    "--file=${Content File Basename}_ipod.m4v",
    "--outfile=ipod_publish_description_file.yaml"
  );
};
```

The `--file` argument to the `publish` task should be the name of the iPod encoded podcast.

Custom Actions

Most of the commands and operations that you need to run within a workflow are handled by the `pcastaction` command. To run custom scripts, such as transformations, additional formatting, or specialized encoding and merging of content, you can create a script within the `Tools` directory of the workflow, and then execute this custom script within one of the actions.

You should still use `pcastaction` as the wrapper, as it helps to ensure that the environment is set correctly. To specify the use of a custom script, use the shell task within `pcastaction`, then specify the commands and arguments. The properties supported by Podcast Producer will not automatically be available to your script, so your script must be able to accept the arguments.

For example, the sample script below generates a graphical version of the title and author of a podcast for use when the item is published. The title and author name are supplied as standard arguments to the script and you could execute it on the command line:

```
$ generate_title.pl --title="${Title}" \
  --author="${User Full Name}" \
  --outfile=title.png
```

You can rewrite the equivalent command line as a script item within the Action Definition File in a workflow:

```
generate_title = {
  dependsOnTasks = (preflight);

  command = "/usr/bin/pcastaction";
  arguments = (
    shell,
    "--",
    "/usr/local/bin/Tools/generate_title.pl",
    "--title=${Title}",
    "--author=${User Full Name}",
    "--outfile=title.png",
  );
};
```

By using a combination of the standard tools and custom scripts within actions you can create quite complex processes.

Error Handling

The Podcast Producer workflow execution identifies errors within the different actions by checking the return value from the scripts or commands that are executed. The standard convention is that a process that completes successfully returns 0, whereas an error returns a non-zero (positive) return code.

In the event of an error during the processing of a workflow, the process will be stopped at the point where the action failed. One of the main reasons you should use the `pcastaction` command as a wrapper for your scripts and processes is because it will identify the return codes and stop the processing appropriately with an error message if there is a problem.

When writing custom scripts to process your podcasts, make sure that the script returns the right error code. In a shell script you do this by calling the `exit` function with the appropriate return code. For example, to return a successful execution, you would use a return code of `exit 0`; for a failure you would use a return code of `exit 1`.

Within Ruby, Perl, Python and other scripting environments, similar mechanisms exist to provide a return code to the caller when execution completes.

Workflow Testing and Validation

There are a number of ways in which you can test a new action. The simplest way is to try running the action command on a sample file. Although this only tests the single action, it can be an effective way of testing that the action will do what you expect. For absolute compliance, you should log in as the `pcastxgrid` user and run the action command that you wish to test.

To test the workflow, the most effective way is to try submitting a podcast to the workflow within Podcast Producer. Before running a full test, you can perform a validation of the workflow to ensure that the configuration and components are in place for the workflow to execute.

To execute a validation, use the `pcastconfig` command line tool:

```
$ pcastconfig --validate_workflow_at_path path_to_workflow
```

If the workflow is already installed within the system or local workflow directories, then you can also validate by name, as shown here:

```
$ pcastconfig --validate_workflow workflow_name
```

Or you can validate all of the workflows within your system, as shown here:

```
$ pcastconfig --validate_all_workflows
```

The command will highlight any potential issues with the structure and content of the workflow.

Advanced Actions

When you submit a job to Podcast Producer, if the server has been configured to make use of Xgrid agents, then the workflow will be distributed to a client for processing. Which client is used depends on a system built into Xgrid called the scoreboard.

The Xgrid client chosen depends on a score generated by the agent using a scoreboard system that ranks the different clients and the level and processing of the work involved in the podcast submission. When a client submits a podcast to Podcast Producer, the Workflow Specification File includes an Agent Ranking Tool (ART), which is a script or executable that can be executed on the client to determine the score and therefore suitability for executing the job. An associated property, `artConditions`, is used to determine what ART value should be returned by an agent when the associated ART script is executed.

When a podcast is submitted to Podcast Producer, the ART script is executed on each XGrid agent and the information returned by this process is then used by the Xgrid manager to determine which agent should be used to handle the basic processing. In the example workflows in this article, the script and property returns 1, but you can tailor this value and determination by modifying the script and property.

For all workflows that you want to integrate with Xgrid execution, you should also ensure that the tools used in the workflow are available on all of the machines in your Xgrid system. In particular, custom scripts and applications that you are using to process your podcasts should either be located within the Tools directory of the Resources directory on the shared directory, or, you can place workflow-specific scripts and tools into the Tools directory of the Xgrid bundle.

Executing Actions in Parallel

Another way to maximize the performance of your workflows and actions during execution is to support multiple concurrent actions. For example, if you are designing a workflow that creates multiple target formats then you can often process each of these formats in parallel, since each action is independent of the others. See Figure 5 for an example of a parallel workflow.

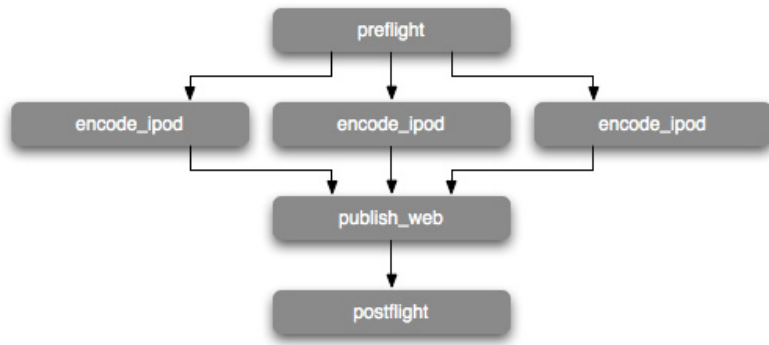


Figure 5: Workflow Action Sequence with parallel tasks

To set up a parallel execution, you specify multiple actions with a common dependency. From the above Figure 5, all three Encode actions have the common preflight dependency. Podcast Producer will identify the ability to run the tasks in parallel automatically for you.

The Xgrid agent will run as many parallel tasks as it can based upon the number of available CPU cores. In the example in Figure 5, three simultaneous target formats have been specified. On a system with four cores, all three of these processes could be executed in parallel. Here you can see a skeleton action matching the process:

```

(
  {
    taskSpecifications = {
      preflight = {
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
      encode_ipod = {
        dependsOnTasks = (preflight);
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
      encode_itunes = {
        dependsOnTasks = (preflight);
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
      encode_iphone = {
        dependsOnTasks = (preflight);
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
      postflight = {
        dependsOnTasks = (encode_iphone,encode_ipod,encode_itunes);
        command = "/usr/bin/pcastaction";
        arguments = (
        );
      };
    };
  }
)

```

You can always monitor the execution of workflows across your Xgrid agents by using the Xgrid Admin Application.

Naming Your Actions Safely

If you are creating a number of different workflows to handle different target formats for your podcasts then you can maximize the flexibility of your workflows and their actions by standardizing on the actions and how they work with each other, so that you can adapt and modify workflows using existing actions.

For example, if you frequently create workflows that encode to a format suitable for the iPod, you can create a generic action that always expects the same input file and always generates an output file with the same name or format. This will allow you to re-use actions in different workflows without having to rename or rebuild the action each time you use it.

Here you can see an example of a hard-coded action that relies upon a specific output filename:

```

encode_audio = {
  dependsOnTasks = (preflight);
}

```

```

command = "/usr/bin/pcastaction";
arguments = (
  encode,
  "--basedir=${Base Directory}",
  "--input=${Content File Basename}${Content File Extension}",
  "--output=${Content File Basename}-audio.m4a",
  "--encoder=mp4_audio_high"
);
};

```

Keep in mind that in the Action Definition File, the action name and the output file is not specific enough to be globally identifiable and therefore it is not reusable within all of your workflows. Within a large workflow, simple filenames like this may introduce processing problems if multiple actions write to the same file. The action should be re-written to describe the output file more precisely:

```

encode_mp4_audio_high = {
  dependsOnTasks = (preflight);
  command = "/usr/bin/pcastaction";
  arguments = (
    encode,
    "--basedir=${Base Directory}",
    "--input=${Content File Basename}${Content File Extension}",
    "--output=${Content File Basename}-mp4_audio_high.m4a",
    "--encoder=mp4_audio_high"
  );
};

```

With a more specific, but still generic name and output file, it should be easier to re-use and deploy your actions in different workflows, which will make them easier to work with and easier to use within parallel processing environments.

Resources and Performance

Actions can require a significant amount of resources, both in terms of the CPU and memory usage, and in the disk resources required to create the various temporary files. You cannot control the CPU and memory usage, or limit it, but you can create a workflow that is designed to make the best of your environment so that it does not cause a significant problem.

Using parallel actions, for example, may not be desirable if some of the actions that you have specified require a large amount of resources to execute. By staggering the execution of actions, and creating workflows that do not require a large number of simultaneous executions, you can easily lower the overall requirements of the process.

Disk storage during the processing of a podcast through a workflow is handled by the Podcast Producer Server and the directory configured by the Shared File System option within the Podcast Producer settings. The entire directory is shared, with the Podcast job directory within that shared location providing the storage space for all operations. This is the directory exposed through the Base Directory property and applied to pcastaction through the `basedir` option.

During the configuration of Podcast Producer you should ensure that the shared directory has enough space to cope with the requirements of your production environment. By default, the postflight script deletes the contents of the workflow task temporary directory once the processing has been completed. In most cases this solution eliminates the problem of the created tasks and associated temporary files taking up space required for future productions. You can change the postflight script so that these temporary files are not deleted if you want to check the execution.

However, in a complex workflow, or a busy system configured with a number of different systems processing podcasts simultaneously, you may want to delete files between stages. The easiest way to do this is to create a new action that uses a custom script that performs the cleanup process.

Conclusion

In this article we have examined a range of issues surrounding the development and definition of actions within Podcast Producer workflows. Actions, on their own, are just the definition of the command lines that will process your podcast. The internal tools (particularly pcastaction), provide the bulk of the functionality for the actions. For custom environments, you can also create your own tools and scripts to be executed as an action.

Chaining the actions together is more complex. It is through the use of the dynamic properties that different stages of the process can be united. These dynamic properties specify the environment unique to a given podcast and they help identify inputs and outputs during processing. We have also seen how a careful action definition can make the distribution of work easier through Xgrid agents by allowing parallel tasks to be executed, and how you can use actions to control resource usage during this processing.

Updated: 2008-11-12