PowerShell Design Specification

|  |  |
| --- | --- |
| monad | PowerShell Cmdlet Designer |

Table of Contents

[1 Introduction 2](#_Toc243303797)

[2 Assumptions & Limitations 2](#_Toc243303798)

[3 External Dependencies 2](#_Toc243303799)

[4 Internal Dependencies 2](#_Toc243303800)

[5 Use Cases 3](#_Toc243303801)

[6 Architectural Diagrams 4](#_Toc243303802)

[7 Implementation Details 5](#_Toc243303803)

[8 Extensibility 21](#_Toc243303804)

[9 Known Issues 21](#_Toc243303805)

PowerShell Design Specification

PowerShell Cmdlet Designer

# Introduction

This document describes architecture and design of the PowerShell Cmdlet Designer. The Cmdlet Designer tool allows users (primarily PMs) to specify all of the metadata required when designing a cmdlet – its name, parameters, validation attributes, and more. Rather than store all of this information in a Word Document, the PowerShell Cmdlet Designer stores its information in a database. By storing all of the spec metadata in a database, these specs now become rich sources of structured information from which we can drive process improvements, and ensure consistency.

# Assumptions & Limitations

* At a high-level, the PowerShell Cmdlet Designer is designed to enforce consistency and prevent name collisions across Microsoft. Enforcing this is a highly-networked operation. In this incarnation, therefore, the tool must be run from a machine connected to a corporate network. See “Extensibility” for more information.

# External Dependencies

* The Cmdlet Designer uses WPF as its interface, meaning that users must have the .NET 3.0 Runtime Components installed: <http://www.microsoft.com/downloads/details.aspx?FamilyId=10CC340B-F857-4A14-83F5-25634C3BF043&displaylang=en>.
* Building the Cmdlet Designer requires the WPF Toolkit to provide support for its DataGrid:  
  <http://wpf.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=29117>.
* The Cmdlet Designer builds on PowerShell, and supports both Version 1 and Version 2, but was driven by support for Version 1. Some of the design oddities (having to run Cmdlet Designer for the first time as an Administrator to install the snapin) can be avoided if Version 2 can be assumed or required. See “Known Issues” for more information.
* See the Deployment Guide for more information.

# Internal Dependencies

* The Cmdlet Designer requires that its supporting web service be deployed and available. See the Deployment Guide for more information.

# Use Cases

* A PM launches the Cmdlet Designer UI. From there, he or she adds nouns, cmdlets, parameters, and parameter set entries.
* The PM sends out a link to the cmdlet for review, which summarizes all of its important characteristics in a read-only fashion.
* A user calls the Cmdlet Designer cmdlets (upon which the UI is built) to script cmdlet-related tasks. For example, they search for all cmdlets that have more than 10 parameters, or update all parameters that have a certain name to use a new name.
* The test team uses the cmdlet comparison tool (also built upon the Cmdlet Designer cmdlets) to verify that the spec for the cmdlet matches its implementation.

# Architectural Diagrams

## Overview

The PowerShell Cmdlet Designer consists of four tiers.

At the lowest level, a database stores the core data for all cmdlets and projects in the enterprise. In order to provide schema flexibility and support for a trusted-subsystem security model, a web service provides a thin interface to the database. This web-service interface is the only public interface: clients cannot connect to the database directly.

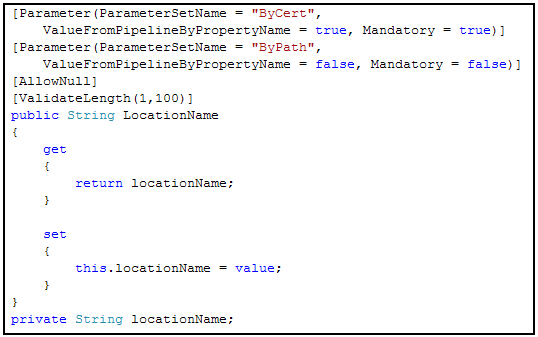
On the client machine, a set of PowerShell cmdlets support the management model of this system and communicate with the web service. The WPF user interface then sits atop these PowerShell cmdlets, and uses them for all interactions with the cmdlet designer back-end.

# Implementation Details

## Overview: Bridging the spec and implementation gap

The primary difficulty in the cmdlet designer system is bridging the gap between spec and implementation.

In the actual cmdlet implementation, you first define .NET Properties. You decorate these properties with validation attributes such as [AllowNull], [ValidateLength(,)], etc. In addition to validation attributes, one attribute (the [Parameter()] attribute) defines this property as a parameter. In that parameter declaration, you define the parameter set name, whether that parameter is mandatory in that parameter set, and more.



This creates a bit of tension with the way we expose cmdlets in help. In help, a cmdlet has parameter sets. Those parameter sets have parameters. Parameters have validation attributes. Help makes no distinction between the attributes attached to the property and the ones defined in the [Parameter] attribute.

Essentially, our user’s mental model of a cmdlet is a SQL join on three aspects of the implementation details: the property name, the validation attributes on the property name, and the entries in the parameter attribute.

To bridge these gaps, we do as much as possible to let users manage parameters and parameter sets without having to be aware of this difference. The “Cmdlet Designer User Interface” section talks more about the steps we take to resolve this tension. As groundwork, we surface the following concepts:

### Parameters

A parameter definition includes the parameter type, parameter name (which is the property name in implementation,) and all validation attributes attached directly to the property (such as AllowNull, ValidateLength, etc.)

### Parameter Set Entries

A parameter set entry definition flows directly from the configuration options available in the [Parameter()] attribute. Parameter set entries define most of the major validation attributes: “Mandatory,” “ValueFromPipeline,” “Parameter Set Name,” etc. This makes it clear that these attributes can vary between parameter sets.

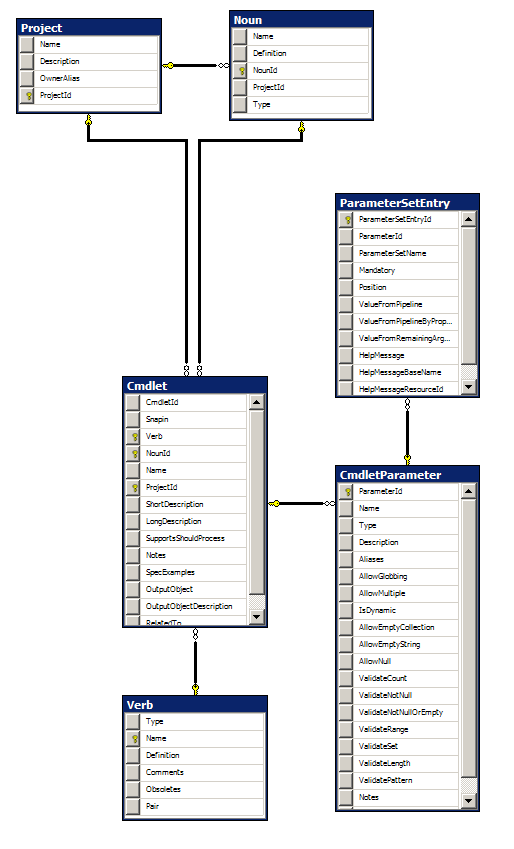
## Cmdlet Designer database

### Table structure

The two main conceptual units in the cmdlet designer database are projects and cmdlets. Project owners manage information about their project explicitly (such as the people allowed to read and write to cmdlets in that project.) Cmdlet authors manage the information about cmdlets both explicitly and implicitly: by directly editing cmdlet details, and by using features of the cmdlet designer that make changes to underlying parameter validation attributes on their behalf.

The CmdletParameter table contains information about the parameters of a cmdlet (using the slightly tweaked definition of “parameter” from above, while the ParameterSetEntry table contains information about parameter set entries.

The database also tracks Verbs, Nouns, and possibly a few other lookup tables. The following diagram illustrates the high-level table structure:



For a full definition of the database schema, attach to a running instance or examine the database setup script.

### Normalization

In general, this database schema normalizes repetitive and disjoint data. There are 12 instances of non-normalized data, though:

Verb types

The verb table includes information about the type of verb: common, data, security, etc. These could be extracted to instead be “VerbTypeId” and a VerbType table. Since this entire table is a hard-coded lookup table scripted from an Excel spec, there is no concern of insert / update / delete anomalies.

Snapin names

Cmdlet specs track information about the snapin in which they are defined. This is a free-form text field, which introduces the risk that a PM might spec the cmdlet to be in a snapin that does not exist, or might mistype a snapin that does exist. To have any impact, guaranteeing data consistency through normalization into a “snapin” table would also imply a work flow to let PMs or project owners manage data in this table. That would be an un-desirable feature from both the user and cost perspective, so snapin names are defined inline.

Output objects

Cmdlet specs track information about the objects they emit. This is a free-form text field, which introduces the risk that a PM might spec the cmdlet to emit an object that does not exist, or might mistype one that does exist.

To have any impact, guaranteeing data consistency through normalization into a “Output Object” table would also imply a work flow to let PMs or project owners manage data in this table. That is a potential future feature, at which time this column would represent an object ID.

Cmdlet Related To

Cmdlet specs track information about closely related cmdlets. This is a free-form text field, which introduces the risk that a PM might refer to a cmdlet that does not exist, or might mistype a snapin that does exist. To have any impact, guaranteeing data consistency through normalization into the cmdlet table would imply that a PM could not put information into the “related to” section without first creating a spec for it. That would be an un-desirable feature from the user perspective. It would also imply a “cmdlet picker” type of functionality to generate the related-to field, which would be un-desirable from a cost perspective.

Parameter types

Parameters contain information about the underlying .NET type they hold. This is a free-form text field, which introduces the risk that a PM might spec the parameter to be of a type that does not exist, or might mistype one that does exist.

Like the “output object” section, guaranteeing data consistency through normalization into a “Parameter type” table would also imply a work flow to let PMs or project owners manage data in this table. That is a potential future feature (through the Object Designer functionality,) at which time this column may represent a type ID.

Parameter aliases

Parameters store information about their possible aliases (such as “PsPath”.) This is a free-form text field, which introduces the risk that a PM might spec the parameter to have an alias that is not desired. To have any impact, guaranteeing data consistency through normalization into a “ParameterAlias” table would also imply a work flow to let PMs or project owners manage data in this table. That would be an un-desirable feature from both the user and cost perspective, so parameter aliases are defined inline.

Parameter set names

Parameter set entries store information about the parameter set in which they are defined. This is a free-form text field, which introduces the risk that a PM might spec the parameter set entry be in a parameter set that is not desired. Because of data duplication, it also introduces the risk of an update anomaly, since renaming a parameter set is not an atomic operation (you would have to update each parameter set entry to use the new parameter set name.)

To have any impact, guaranteeing data consistency through normalization into a “ParameterSetName” table would also imply a work flow to let PMs or project owners explicitly manage data in this table. That would be an un-desirable feature from both the user and cost perspective, so parameter set names are defined inline.

It might be argued that the parameter-level parameter set editing user interface control in fact offers a way to manage these parameter sets. Since the control offers no validation of parameter set names, though, implicit management of a ParameterSetName table by this control would be the same as managing the underlying data directly.

Parameter set help message base names / help message resource ids

Parameters set entries store information about their help message base name and help message resource id. This is a free-form text field, which introduces the risk that a PM might spec the parameter set entry to have data that is not desired. To have any impact, guaranteeing data consistency through normalization into a “HelpMessageBaseName” or “HelpMessageResourceId” table would also imply a work flow to let PMs or project owners manage data in this table. That would be an un-desirable feature from both the user and cost perspective, so parameter aliases are defined inline.

### History

History tracking is integral to the Cmdlet Designer database. Each table that contains live data (which is all tables but the Verb table) contains a history tracking table that shares the same definition. In addition to the columns of the original table, the history table also tracks the usename of the person who implemented the change.

For example, where we have the Cmdlet table, we also have the CmdletHistory table. For every change that we make to a live table, we add an entry to the appropriate history table. This history table has no constraints (foreign key, data integrity, or otherwise,) so that the live data can change with impunity.

### Security and Stored Procedures

All database access goes through stored procedures. The stored procedures enforce all data validation, business rules, and integrity constraints. This gives us schema flexibility for the underlying table structure, but most importantly, enables a Trusted Subsystem model. The web service is the only public interface to the back-end data store. When it calls these stored procedures, it also supplies the logon name of the currently authenticated user. The stored procedures then use this logon information to track auditing information, or to check permission to modify data.

Through its trusted-subsystem model, the database and tool chain implement a role-based security mechanism. Cmdlets let you manage the users that have access to your project, and their role (Project Owner, Spec Writer.)

|  |  |
| --- | --- |
| Cmdlet | Minimum Role |
| New-Cmdlet | SpecWriter |
| New-Noun | ProjectOwner |
| New-Parameter | SpecWriter |
| New-ParameterSetEntry | SpecWriter |
| New-ProjectUser | ProjectOwner |
| Remove-Cmdlet | SpecWriter |
| Remove-Noun | ProjectOwner |
| Remove-Parameter | SpecWriter |
| Remove-ParameterSetEntry | SpecWriter |
| Remove-Project | ProjectOwner |
| Remove-ProjectUser | ProjectOwner |
| Update-Cmdlet | SpecWriter |
| Update-Noun | ProjectOwner |
| Update-Parameter | SpecWriter |
| Update-ParameterSetEntry | SpecWriter |
| Update-Project | ProjectOwner |
| <All Remaining> | No permission required |

### Deployment

The Cmdlet Designer database works on both SQL Express databases, and full-fledged SQL Server databases. This is to ensure that the Cmdlet Designer supports a low-scale environment the same as it would in a production one.

See the included Deployment Guide for detailed setup information.

## Cmdlet Designer web service

### Overview

The cmdlet designer web service provides the public entry-point for the underlying database. Its setter methods almost exactly mimic the database stored procedures. Its getter methods also mimic the database stored procedures, but it also wrap the raw row results into the structured objects (Project, Cmdlet, etc) they represent.

### Implementation patterns

All web service methods follow a consistent pattern. Each defines and populates a dictionary of stored procedure parameters, and then provides that dictionary as a parameter to the SqlUtils.ExecuteStoredProcedure method.

The SqlUtils.ExecuteStoredProcedure method then creates a SqlCommand for that stored procedure and executes it.

If the stored procedure returns information, an override to the SqlUtils.ExecuteStoredProcedure method lets you provide a generic collection into which the method places the results.

The datatype of items in this generic collection (such as AspenCmdlet, AspenProject, etc) define how the stored procedure converts its results into strongly typed objects. Each type must inherit from IAspenDataReader. The IAspenDataReader interface defines a method, void Initialize(SqlDataReader reader), which the ExecuteStoredProcedure method calls with the current row being retrieved from the database. The AspenCmdlet class (for example,) then defines how it populates its instance fields from a database row result.

### Security

Since the web service and database together implement a trusted subsystem model, all database stored procedures require a @user parameter. To provide this, all web service methods retrieve the user identity before the web service call, and add that as a parameter to the stored procedure.

### Deployment

Deployment of this web service requires that the DataSource server name in the ConnectionString web.config setting be updated to point at an appropriate server. The rest of settings (initial catalog and authentication) must match the settings used in the database creatiion script.

In order to support the security model, anonymous authentication must be disabled in the IIS settings of the web service.

## Cmdlet Designer cmdlets

### Overview

The cmdlet designer cmdlets form the interface to all functionality in the cmdlet designer. While the web service is technically the lowest level of the stack, its sole purpose is to define a security boundary that we can protect. The cmdlets, on the other hand, represent both the public API and all management functionality.

### Data Mapping

Aside from some minor usability checks and validation, most of the cmdlet functionality focuses on interacting with the cmdlet designer web service, and surfacing objects that represent the management functionality: Projects, Cmdlets, and more.

While it would be possible for the cmdlets to directly surface the objects returned by the web service, the cmdlet project instead defines new classes for each. This decouples the cmdlet consumers from the underlying web service, and allows additional functionality (such as cloning) on the objects that the cmdlets produce.

In general, the pattern that supports this is simple. As the cmdlets retrieve the strongly structured data from the web service, create new objects using a constructor that takes all required data:

foreach (AspenCmdlets.AspenCmdletService.AspenCmdlet cmdlet in cmdlets)

{

WriteObject(new AspenCmdlet(

cmdlet.Id, cmdlet.Snapin, cmdlet.Verb, cmdlet.Noun, cmdlet.Name,

cmdlet.Project, cmdlet.ShortDescription, cmdlet.LongDescription,

cmdlet.SupportsShouldProcess, cmdlet.Notes,

cmdlet.SpecExamples, cmdlet.OutputObject, cmdlet.OutputObjectDescription,

cmdlet.RelatedTo));

}

### Installation and Deployment

The Aspen Management cmdlets sit inside of a single snapin. You can run the Install-AspenCmdletSnapin.ps1 script to install these cmdlets, or call InstallUtil.exe directly.

Since all cmdlets speak with the cmdlet designer web service back-end, it is important that they allow you to work with different web services. By default, they point to the production URI, but you can override that setting with the –ServiceUri common parameter.

### Security

The cmdlet designer cmdlets do not constitute a security boundary. While they provide usability benefits (such as prompting before deleting data,) security is ultimately handled between the web service and database stored procedures.

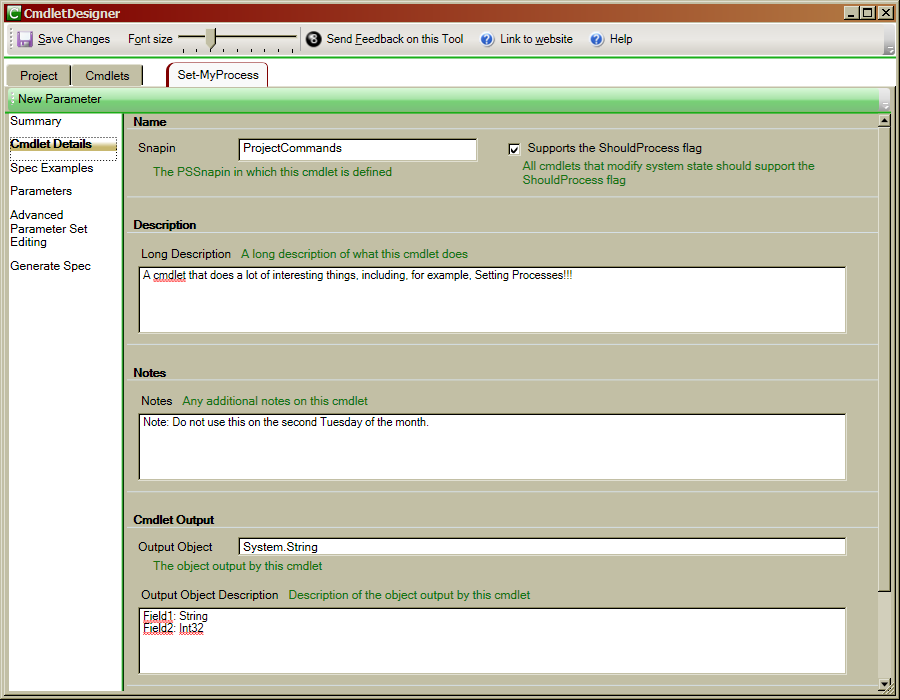
## Cmdlet Designer user interface

The Cmdlet Designer builds on WPF as its user interface technology.

### Selection of User Interface Technology

### Overview

The Cmdlet Designer UI provides an interface to the cmdlets, web service, and database back-end that stores all cmdlet metadata at Microsoft.



It uses both of PowerShell’s automation techniques to accomplish UI-on-cmdlets structure. For commands that use only fully-specified parameters, the Cmdlet Designer uses PowerShell’s task model:

// Get the cmdlets for a project

private List<AspenCmdlet> GetCmdlets(string projectName)

{

GetCmdletCommand command = new GetCmdletCommand();

command.ServiceUri = ServiceUri;

command.ProjectName = projectName;

List<AspenCmdlet> results = new List<AspenCmdlet>();

try

{

foreach (AspenCmdlet result in command.Invoke())

{

results.Add(result);

}

}

catch (Exception exception)

{

MessageBox.Show(exception.Message);

}

return results;

}

For those that modify data, the Cmdlet Designer uses PowerShell’s runspace model. This makes it much more clear to call the target commands —which all take their information from properties that support ValueFromPipelineByPropertyName:

// Process the updates

foreach(AspenCmdletParameterSetEntry currentUpdatedParameter in updatedEntries)

{

using (Pipeline pipeline = runspace.CreatePipeline())

{

Command updateCommand = new Command("Set-CmdletParameterSetEntry");

updateCommand.Parameters.Add("ServiceUri", ServiceUri);

pipeline.Input.Write(currentUpdatedParameter);

pipeline.Commands.Add(updateCommand);

Collection<PSObject> results = pipeline.Invoke();

HandleErrors(pipeline);

}

}

### Data Binding

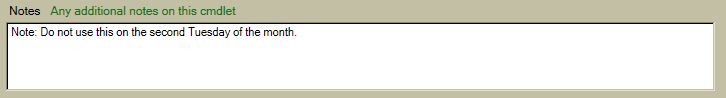
The user interface heavily leverages WPF’s data binding philosophy. When you load a cmdlet, the Cmdlet Designer binds its fields onto custom controls that represent summary information, details, spec examples, parameters, and advanced parameter set information. These custom controls are either visible or hidden, depending on the navigation header you’ve selected from the left-hand pane.

In addition to these large-scale controls, lower-level controls form the basis of data entry and display. Since we want help text to be ubiquitous and contextual, controls exist to display this information in a consistent and uniform way:

ExtendedTextBox



FreeTextExtendedTextBox



ExtendedCheckBox



ExtendedListBox



These controls support data binding of their appropriate data type.

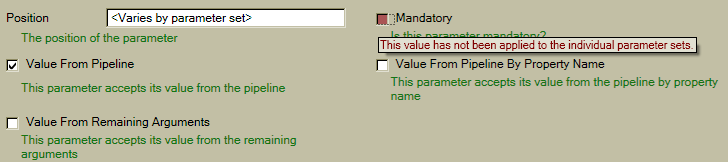
### Parameter Set Editing

As mentioned in the introduction, the majority of the complexity in the Cmdlet Designer comes from bridging the conceptual gap between spec and implementation. During specification, users think of parameters that have a name, major validation attributes (i.e.: Mandatory, ValueFromPipelineByProperyName,) minor validation attributes (i.e.: ValidateLength, ValidatePattern,) and exist as part of one or more parameter sets.

What they don’t immediately realize (but often require,) is that PowerShell allows you to change the major validation attributes between parameter sets. For example, a parameter can be mandatory in one parameter set, but not in another. In implementation, you represent that as a single parameter object with multiple associated ParameterSetEntry objects.

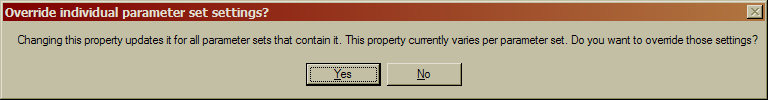
To resolve this, the AspenCmdletParameterDetails class lets you manage these details at the parameter level. This class surfaces a union of the individual parameter with all of its associated parameter set entries — a single object with name, major validation attributes, minor validation attributes, and parameter set names.

#### Validation Attributes



If all of the underlying ParameterSetEntry objects share the same value for a given attribute (such as Mandatory,) the AspenCmdletParameterDetails class surfaces that value directly as an aggregate. Changing this aggregate value transparently changes the value for all of the underlying ParameterSetEntry objects, as well. For example, if all underlying ParameterSetEntry objects have Mandatory=False, the Mandatory property has the value “False.” If you change this to “True,” the AspenCmdletParameterDetails class applies that value to all underlying ParameterSetEntry objects.

If all underlying objects do not share the same value, however, the AspenCmdletParameterDetails class surfaces a representation of indeterminate: null for any of the value types, or “Varies by Parameter Set” for a string property. Changing an indeterminate property first provides a warning, but applies the value to all underlying ParameterSetEntry objects if you choose to continue.



#### Parameter Sets

In addition to validation attributes, users want to easily select which parameter sets a parameter is part of, add parameters to new parameter sets, and remove parameters from parameter sets.



The AspenCmdletParameterDetails class supports this as well. For a given parameter (and set of underlying ParameterSetEntry objects,) the class surfaces a List<String> of parameter set names.

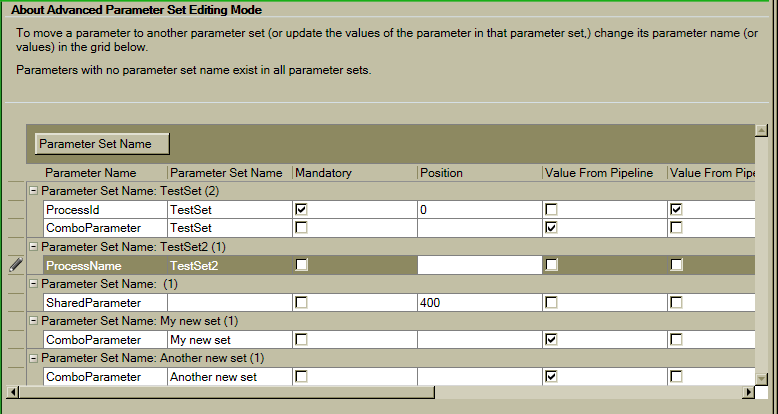
Unchecking a parameter set name removes the ParameterSetEntry that corresponds to that ParameterSetName from the underlying collection of ParameterSetEntry objects. It doesn’t do this permanently, though. It stores these removed entries in an internal list, so that the user can add the parameter to the parameter set again without losing data.

Adding a new parameter set name (or selecting a checkbox of an un-checked parameter set name) adds a new ParameterSetEntry for that parameter. If a ParameterSetEntry for that ParameterSet name exists in the internal recycle bin, this action restores all of its previous values. If it does not exist in the internal recycle bin, this new ParameterSetEntry takes its defaults from the corresponding aggregate property. If the aggregate value is indeterminate, it uses the default for that attribute.

The list of parameter set names comes from the combination of all parameters on the cmdlet (not just on that parameter,) so that users can easily add a parameter to another parameter set.

### Advanced Parameter Set Editing

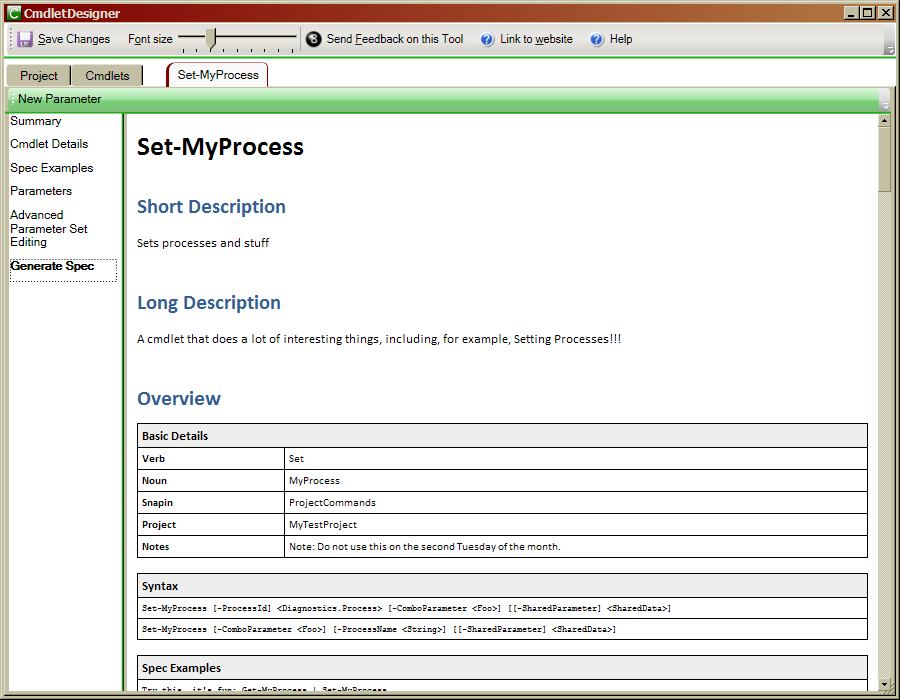
The advanced parameter set editing control allows users to modify the actual parameter set entries used by the parameters in their cmdlet.



In this mode, they can group, sort, and modify data for any of the parameter set entries. This is also the only way to modify details that we surface as an aggregate in the parameter set editing page.

### Generate Spec

The Generate Spec page provides a read-only spec view of the cmdlet. This allows users to see a version more suitable for review. From this view, they can also copy and paste the content into Word to create a version that they can use for change tracking and reviewer comments.



The content of this view is generated by the Get-CmdletDesignerSpecHtml.ps1 script. Implementing this document view as a script allows tool users to see how to use the cmdlets, and how they might write scripts to automate their own management or documentation needs.

### Installation / Deployment

The only installation requirement of the tool is the registration of the Aspen Management cmdlets snapin. You can run the Install-AspenCmdletSnapin.ps1 script to install this snapin, or call InstallUtil.exe directly. Once the snapin is registered, the tool (and supporting cmdlet DLL) may be upgraded in place simply by replacing the files with new versions.

Since all cmdlets speak with the cmdlet designer web service back-end, the tool allows you to work with different web services. By default, it points to the hard-coded production service URI, but you can override that setting by modifying the CmdletDesigner.exe.config file:

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<appSettings>

<add key="ServiceUri" value="http://devserver/cmdlet.asmx"/>

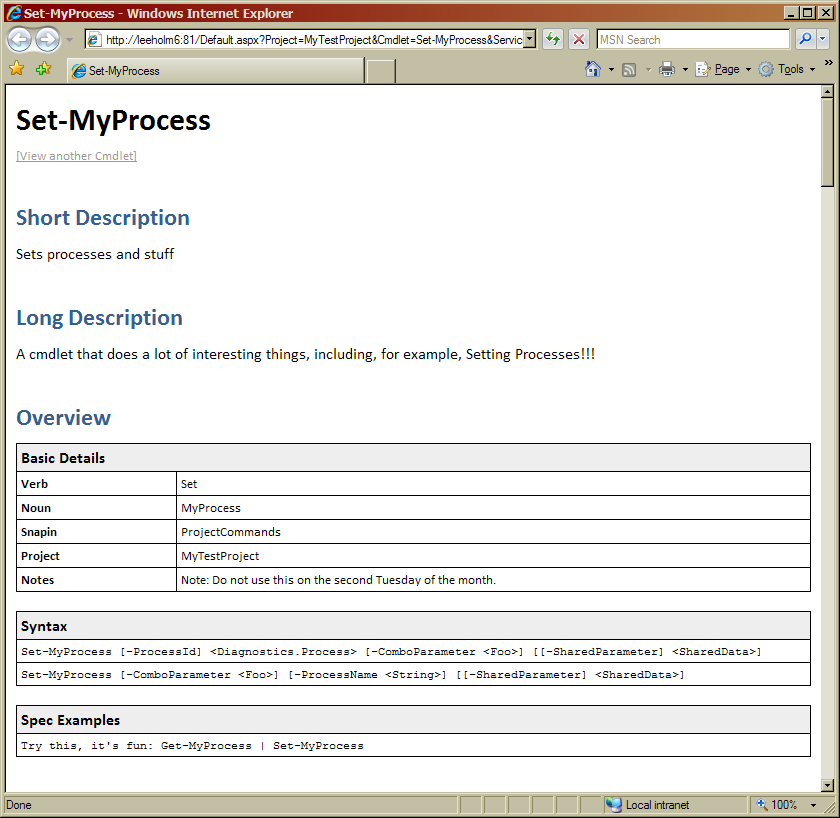
</appSettings>

</configuration>

## Cmdlet Designer Spec Viewer

### Overview

The Cmdlet Designer Spec Viewer provides a read-only, ASP.Net-based view of the cmdlets in a project. This allows you to share spec links as URLs, and lets you review specs without requiring that the tool be installed.



This ASP.Net viewer uses almost exactly the same script as the one used in the Cmdlet Designer UI, except for the minor addition to *[View another cmdlet]*.

If you visit the page without any parameters, it asks you to select a project. After you select a project, you can select a cmdlet. The site uses querystring parameters exclusively so that users can easily share spec URLs.

### Deployment

Since all cmdlets speak with the cmdlet designer web service back-end, the site allows you to work with different web services. By default, it points to the production service URI, but you can override that setting by providing a ServiceUri querystring parameter. This parameter is restricted to a hard-coded list defined in web.config to prevent people from influencing the spec viewer to connect to random network endpoints.

# Extensibility

## Non-networked operation

A core requirement for the Cmdlet Designer is detecting Microsoft-wide name collisions. This naturally enforces a database / networked requirement. When you add the requirement for permission management (preventing people from accidentally damaging other projects,) user authentication becomes another requirement. The Cmdlet Designer relies on Windows Authentication to support this.

To enable non-networked operation, the natural architectural boundary is the cmdlets. It would be fairly straight-forward to write new versions of the cmdlets that act the same as the webservice-based ones, yet instead fetch data from and store data in an XML or CSV file. The file-based cmdlets could emit the same output objects as the webservice-based ones do, so changes to the UI would be minor.

# Known Issues

## Snapin installation

The Cmdlet Designer works with both PowerShell V1 and PowerShell V2. In order to support PowerShell V1, the underlying cmdlets are developed as a Snapin. Snapins require explicit installation, causing frequent pain for users of its ClickOnce deployment. ClickOnce does not support an administrative installation stage, so the Cmdlet Designer detects when the snapin is not installed. It then tries to install it. If this is unsuccessful, it generates a dialog box telling users how to relaunch the Cmdlet Designer as an administrator. If they copy and paste those directions, the Cmdlet Designer is then able to install the snapin. Despite this verbose error message, this is the most common usability problem in the Cmdlet Designer. If PowerShell V2 can be assumed, loading the snapin instead as a binary module would eliminate this altogether.

## Execution policy

Some portions of the Cmdlet Designer are implemented as scripts – the generated specs, help MAML, and C# source code. These fail when the user’s Execution Policy is set to Restricted. Although not implemented, the Cmdlet Designer could set the PSExecutionPolicyPreference environment variable to “RemoteSigned” to have PowerShell V2 trust these local scripts for the current process only.