

Management Pack Authoring Guide

Microsoft Corporation

Published: April 16, 2010

Author

Brian Wren

Feedback

Send suggestions and comments about this document to [momdocs@microsoft.com](mailto:momdocs@microsoft.com?subject=MPGuide). Please include the guide name and published date with your feedback.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, Active Directory, ActiveSync, Internet Explorer, JScript, SharePoint, SQL Server, Visio, Visual Basic, Visual Studio, Win32, Windows, Windows PowerShell, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Revision History

|  |  |
| --- | --- |
| Release Date | Changes |
| January 7, 2010 | * Initial release of this guide including only the Service Model section. |
| January 29, 2010 | * Complete Composition section added. * Basic structure and introduction to remaining sections added. These sections are to be completed in subsequent versions. |
| April 16, 2010 | * Complete Health Model section added. * Corrections made in How To topics in Service Model and Composition sections. * Corrections made in Recommended Base Classes table. |

Contents

[Management Pack Authoring Guide 16](#_Toc260754934)

[In This Section 16](#_Toc260754935)

[Management Pack Basics 17](#_Toc260754936)

[In This Section 17](#_Toc260754937)

[What is a management pack? 17](#_Toc260754938)

[Editing Tools 18](#_Toc260754939)

[Naming 19](#_Toc260754940)

[References 20](#_Toc260754941)

[Service Model 20](#_Toc260754942)

[In This Section 20](#_Toc260754943)

[Key Concepts 21](#_Toc260754944)

[In This Section 21](#_Toc260754945)

[Classes 21](#_Toc260754946)

[Properties 22](#_Toc260754947)

[Key Properties 22](#_Toc260754948)

[Base Classes and Inheritance 23](#_Toc260754949)

[Class Types 25](#_Toc260754950)

[Abstract Classes 25](#_Toc260754951)

[Singleton Classes 25](#_Toc260754952)

[See Also 26](#_Toc260754953)

[Relationships 26](#_Toc260754954)

[Hosting Relationship Type 26](#_Toc260754955)

[Containment Relationship Type 30](#_Toc260754956)

[Reference Relationship Type 31](#_Toc260754957)

[See Also 31](#_Toc260754958)

[Viewing Classes in the Consoles 31](#_Toc260754959)

[Prerequisites 31](#_Toc260754960)

[See Also 33](#_Toc260754961)

[Discovery 33](#_Toc260754962)

[The Mechanics of Discovery 33](#_Toc260754963)

[Discovery Data 35](#_Toc260754964)

[Discovery Data Sources 36](#_Toc260754965)

[Registry 36](#_Toc260754966)

[WMI 36](#_Toc260754967)

[Scripts 36](#_Toc260754968)

[Discovery Targets 37](#_Toc260754969)

[Discovering Relationships 37](#_Toc260754970)

[Hosting Relationships 37](#_Toc260754971)

[Containment Relationships 38](#_Toc260754972)

[Populating Groups 39](#_Toc260754973)

[Proxy Discoveries 39](#_Toc260754974)

[Discovery on Demand 39](#_Toc260754975)

[See Also 40](#_Toc260754976)

[Discovery Scripts 40](#_Toc260754977)

[Creating Discovery Scripts 40](#_Toc260754978)

[Required Arguments 40](#_Toc260754979)

[MPElement Variables 41](#_Toc260754980)

[Basic Script Structure 41](#_Toc260754981)

[Script Breakdown 42](#_Toc260754982)

[Windows PowerShell 43](#_Toc260754983)

[Empty Discovery Data 44](#_Toc260754984)

[See Also 45](#_Toc260754985)

[Service Model Design Overview 45](#_Toc260754986)

[In This Section 45](#_Toc260754987)

[Defining Classes and Relationships 46](#_Toc260754988)

[Define Application Roles 46](#_Toc260754989)

[Define Application Components for Each Application Role 48](#_Toc260754990)

[Identify Dependencies Between Classes 49](#_Toc260754991)

[Define Application Class 50](#_Toc260754992)

[Groups 51](#_Toc260754993)

[Health Rollups 51](#_Toc260754994)

[Instance Groups 52](#_Toc260754995)

[Computer Groups 52](#_Toc260754996)

[See Also 53](#_Toc260754997)

[Choosing a Base Class 53](#_Toc260754998)

[Criteria for Selecting a Base Class 53](#_Toc260754999)

[Inheritance 53](#_Toc260755000)

[Properties 54](#_Toc260755001)

[Relationships 54](#_Toc260755002)

[Monitoring 55](#_Toc260755003)

[Logical Grouping 55](#_Toc260755004)

[Side Effects 55](#_Toc260755005)

[Custom Base Classes 56](#_Toc260755006)

[See Also 57](#_Toc260755007)

[Recommended Base Classes 57](#_Toc260755008)

[Defining Discoveries 59](#_Toc260755009)

[Identifying Discoveries for Each Class 59](#_Toc260755010)

[Basic Discovery Strategy 60](#_Toc260755011)

[Application Roles 60](#_Toc260755012)

[Application Components 60](#_Toc260755013)

[Group Population 61](#_Toc260755014)

[Discovery Intervals 61](#_Toc260755015)

[See Also 62](#_Toc260755016)

[Practices to Avoid 62](#_Toc260755017)

[Classes 62](#_Toc260755018)

[Creating Too Many Classes 62](#_Toc260755019)

[Creating Classes for Volatile Objects 63](#_Toc260755020)

[Properties that Update Too Frequently 63](#_Toc260755021)

[Discoveries 64](#_Toc260755022)

[Too Frequent Discoveries 64](#_Toc260755023)

[Script Discoveries Targeting Broad Classes 64](#_Toc260755024)

[See Also 64](#_Toc260755025)

[Building a Service Model 64](#_Toc260755026)

[In This Section 65](#_Toc260755027)

[Creating Classes and Relationships 65](#_Toc260755028)

[In This Section 65](#_Toc260755029)

[How to Create a New Management Pack 66](#_Toc260755030)

[How to Create a Class 66](#_Toc260755031)

[See Also 68](#_Toc260755032)

[How to Create a Group 68](#_Toc260755033)

[See Also 69](#_Toc260755034)

[How to Create a Containment Relationship 70](#_Toc260755035)

[See Also 71](#_Toc260755036)

[Creating Discoveries 71](#_Toc260755037)

[In This Section 71](#_Toc260755038)

[How to Create a Registry Discovery 72](#_Toc260755039)

[How to Create a WMI Discovery 74](#_Toc260755040)

[See Also 75](#_Toc260755041)

[How to Create a Script Discovery 75](#_Toc260755042)

[How to Create a Windows PowerShell Discovery 77](#_Toc260755043)

[See Also 80](#_Toc260755044)

[How to Populate a Group 81](#_Toc260755045)

[Health Model 83](#_Toc260755046)

[In This Section 83](#_Toc260755047)

[Key Concepts 84](#_Toc260755048)

[In This Section 84](#_Toc260755049)

[Targets 85](#_Toc260755050)

[Effects of a Target 85](#_Toc260755051)

[Which agents the management pack is delivered to 85](#_Toc260755052)

[Which agents the workflow will run on and how many copies of it will be loaded 85](#_Toc260755053)

[Which target variables are available to the workflow 87](#_Toc260755054)

[Which object health state, alerts, and collected data will be associated with 87](#_Toc260755055)

[Targeting Groups 88](#_Toc260755056)

[Data Sources 88](#_Toc260755057)

[See Also 89](#_Toc260755058)

[Data Variables 89](#_Toc260755059)

[Events 91](#_Toc260755060)

[Windows Events 91](#_Toc260755061)

[Criteria 91](#_Toc260755062)

[Properties 92](#_Toc260755063)

[Workflows Supported 93](#_Toc260755064)

[Text Logs 93](#_Toc260755065)

[Properties 93](#_Toc260755066)

[Workflows Supported 94](#_Toc260755067)

[WMI Events 94](#_Toc260755068)

[Criteria 95](#_Toc260755069)

[Properties 95](#_Toc260755070)

[Workflows Supported 96](#_Toc260755071)

[Syslog Events 96](#_Toc260755072)

[Properties 96](#_Toc260755073)

[Facility Values 97](#_Toc260755074)

[Workflows Supported 98](#_Toc260755075)

[Performance Data 98](#_Toc260755076)

[Windows Performance 99](#_Toc260755077)

[Properties 99](#_Toc260755078)

[Workflows Supported 99](#_Toc260755079)

[WMI Performance 99](#_Toc260755080)

[Properties 100](#_Toc260755081)

[Workflows Supported 100](#_Toc260755082)

[See Also 100](#_Toc260755083)

[Monitoring Scripts 100](#_Toc260755084)

[Property Bags 101](#_Toc260755085)

[Typed Property Bags 101](#_Toc260755086)

[Basic Script Structure 102](#_Toc260755087)

[Script Breakdown 102](#_Toc260755088)

[Defining a Script 103](#_Toc260755089)

[Script Name 103](#_Toc260755090)

[Script Arguments 104](#_Toc260755091)

[Script Timeout 104](#_Toc260755092)

[Windows PowerShell 104](#_Toc260755093)

[Supported Workflows 105](#_Toc260755094)

[See Also 106](#_Toc260755095)

[Monitors and Health State 106](#_Toc260755096)

[In This Section 107](#_Toc260755097)

[Unit Monitors 108](#_Toc260755098)

[In This Section 108](#_Toc260755099)

[Event Monitors 109](#_Toc260755100)

[Detection Logic 109](#_Toc260755101)

[Simple Event 109](#_Toc260755102)

[Repeated Events 110](#_Toc260755103)

[Trigger on Timer 110](#_Toc260755104)

[Trigger on Count 110](#_Toc260755105)

[Trigger on Count, Sliding 110](#_Toc260755106)

[Repeated Events Example 111](#_Toc260755107)

[Properties 112](#_Toc260755108)

[Correlated Events 112](#_Toc260755109)

[Correlated Events Example 113](#_Toc260755110)

[Correlated Missing Events 114](#_Toc260755111)

[Correlated Missing Events Example 114](#_Toc260755112)

[Missing Event 115](#_Toc260755113)

[Missing Event Example 116](#_Toc260755114)

[Health Reset Logic 116](#_Toc260755115)

[Event Reset 117](#_Toc260755116)

[Manual reset 117](#_Toc260755117)

[Timer Reset 117](#_Toc260755118)

[Performance Monitors 118](#_Toc260755119)

[Threshold Types 118](#_Toc260755120)

[Simple Threshold 118](#_Toc260755121)

[Double Threshold 119](#_Toc260755122)

[Average Threshold 120](#_Toc260755123)

[Consecutive Samples 121](#_Toc260755124)

[Delta Threshold 121](#_Toc260755125)

[Self-tuning Threshold Monitors 122](#_Toc260755126)

[Script Monitors 122](#_Toc260755127)

[See Also 123](#_Toc260755128)

[Service Monitors 123](#_Toc260755129)

[Aggregate Monitors 123](#_Toc260755130)

[Standard Aggregate Monitors 123](#_Toc260755131)

[Custom Aggregate Monitors 124](#_Toc260755132)

[Health Rollup Policy 125](#_Toc260755133)

[Worst state 125](#_Toc260755134)

[Best state 125](#_Toc260755135)

[Dependency Monitors 126](#_Toc260755136)

[Health Rollup Policy 127](#_Toc260755137)

[Worst state policy 128](#_Toc260755138)

[Best state policy 128](#_Toc260755139)

[Percentage policy 129](#_Toc260755140)

[Health Rollup between Agents 129](#_Toc260755141)

[Alerts from Monitors 130](#_Toc260755142)

[Alert Name 130](#_Toc260755143)

[Alert Description 130](#_Toc260755144)

[Priority and Severity 131](#_Toc260755145)

[Alert Suppression 132](#_Toc260755146)

[Automatic Alert Resolution 132](#_Toc260755147)

[See Also 132](#_Toc260755148)

[Diagnostics and Recoveries 132](#_Toc260755149)

[Diagnostics 132](#_Toc260755150)

[Recoveries 133](#_Toc260755151)

[Recalculating State 134](#_Toc260755152)

[When Diagnostics and Recoveries Run 134](#_Toc260755153)

[Modules 134](#_Toc260755154)

[Rules 135](#_Toc260755155)

[In This Section 135](#_Toc260755156)

[Alert Rules 136](#_Toc260755157)

[Event Alert Rules 136](#_Toc260755158)

[Performance Alert Rules 136](#_Toc260755159)

[Scripting Alert Rules 137](#_Toc260755160)

[Alerts from Rules 137](#_Toc260755161)

[Alert Name 137](#_Toc260755162)

[Alert Description 137](#_Toc260755163)

[Priority and Severity 138](#_Toc260755164)

[Alert Suppression 138](#_Toc260755165)

[Automatic Alert Resolution 139](#_Toc260755166)

[Collection Rules 139](#_Toc260755167)

[Collection rules described 139](#_Toc260755168)

[Event Collection Rules 139](#_Toc260755169)

[Script Based Event Collection 139](#_Toc260755170)

[See Also 140](#_Toc260755171)

[Performance Collection Rules 140](#_Toc260755172)

[Script Based Performance Collection 140](#_Toc260755173)

[Optimized Collection 141](#_Toc260755174)

[See Also 142](#_Toc260755175)

[Tasks 142](#_Toc260755176)

[In This Section 142](#_Toc260755177)

[Console Tasks 142](#_Toc260755178)

[Agent Tasks 143](#_Toc260755179)

[Credentials 144](#_Toc260755180)

[Output 144](#_Toc260755181)

[Kinds of agent tasks 144](#_Toc260755182)

[Health Model Design Overview 144](#_Toc260755183)

[In This Section 144](#_Toc260755184)

[Failure Mode Analysis 145](#_Toc260755185)

[Basic Concept 145](#_Toc260755186)

[Steps in Failure Mode Analysis 146](#_Toc260755187)

[List what can go wrong 146](#_Toc260755188)

[Identify a detection strategy for each failure mode 146](#_Toc260755189)

[Add detection elements to application code 146](#_Toc260755190)

[Plan management pack content 146](#_Toc260755191)

[Operational failures 147](#_Toc260755192)

[Defining Monitors 148](#_Toc260755193)

[Define Unit Monitors 149](#_Toc260755194)

[Diagnostics and Recoveries 151](#_Toc260755195)

[Define Aggregate Monitors 152](#_Toc260755196)

[Define Dependency Monitors 152](#_Toc260755197)

[Hosted Classes 152](#_Toc260755198)

[Health Rollups 152](#_Toc260755199)

[Defining Rules 153](#_Toc260755200)

[Alerting Rules 153](#_Toc260755201)

[Minimizing Noise 153](#_Toc260755202)

[Collection Rules 154](#_Toc260755203)

[See Also 154](#_Toc260755204)

[Practices to Avoid 154](#_Toc260755205)

[Focus on the easy–to-monitor instead of the required 154](#_Toc260755206)

[Running monitors and rules too frequently 154](#_Toc260755207)

[Multiple scripts running at the same time 155](#_Toc260755208)

[Targeting classes with multiple instances 155](#_Toc260755209)

[See Also 155](#_Toc260755210)

[Building a Health Model 155](#_Toc260755211)

[In This Section 156](#_Toc260755212)

[Creating Monitors 156](#_Toc260755213)

[In This Section 156](#_Toc260755214)

[How to create an event monitor 157](#_Toc260755215)

[See Also 158](#_Toc260755216)

[How to create an event monitor targeted at class with multiple instances 158](#_Toc260755217)

[See Also 160](#_Toc260755218)

[How to create a repeating event monitor 160](#_Toc260755219)

[See Also 161](#_Toc260755220)

[How to create a monitor based on a script 161](#_Toc260755221)

[See Also 163](#_Toc260755222)

[How to create a WMI event monitor 164](#_Toc260755223)

[See Also 165](#_Toc260755224)

[How to create a consecutive sample performance monitor 166](#_Toc260755225)

[See Also 167](#_Toc260755226)

[How to create a dependency monitor 167](#_Toc260755227)

[See Also 168](#_Toc260755228)

[Creating Diagnostics and Recoveries 169](#_Toc260755229)

[In This Section 169](#_Toc260755230)

[How to create a diagnostic 169](#_Toc260755231)

[See Also 170](#_Toc260755232)

[How to create a recovery based on the output of a diagnostic 170](#_Toc260755233)

[See Also 174](#_Toc260755234)

[Creating Rules 174](#_Toc260755235)

[In This Section 174](#_Toc260755236)

[How to create a delimited text log alerting rule 174](#_Toc260755237)

[See Also 176](#_Toc260755238)

[How to create a WMI performance collection rule 176](#_Toc260755239)

[See Also 177](#_Toc260755240)

[How to create a script-based performance collection rule 177](#_Toc260755241)

[See Also 179](#_Toc260755242)

[How to create a script-based event collection rule 179](#_Toc260755243)

[See Also 180](#_Toc260755244)

[Presentation 181](#_Toc260755245)

[In This Section 181](#_Toc260755246)

[Views 181](#_Toc260755247)

[Product Knowledge 182](#_Toc260755248)

[Composition 182](#_Toc260755249)

[In This Section 182](#_Toc260755250)

[Key Concepts 183](#_Toc260755251)

[In This Section 183](#_Toc260755252)

[Module and Workflow Basics 184](#_Toc260755253)

[Modules 184](#_Toc260755254)

[Module Parameters 185](#_Toc260755255)

[Data Stream 186](#_Toc260755256)

[Data Types 186](#_Toc260755257)

[Workflows 186](#_Toc260755258)

[Workflow Targets 187](#_Toc260755259)

[Modules Types 188](#_Toc260755260)

[In This Section 188](#_Toc260755261)

[Data Source Modules 188](#_Toc260755262)

[Common Data Source Modules 189](#_Toc260755263)

[Probe Action Modules 193](#_Toc260755264)

[Common Probe Action Modules 194](#_Toc260755265)

[Condition Detection Modules 196](#_Toc260755266)

[Common Condition Detection Modules 197](#_Toc260755267)

[Write Action Modules 198](#_Toc260755268)

[Common Write Action Modules 198](#_Toc260755269)

[Module Implementations 200](#_Toc260755270)

[Composite Module Example 201](#_Toc260755271)

[Sample Workflow with Composite Modules 202](#_Toc260755272)

[Kinds of Workflows 203](#_Toc260755273)

[Discoveries 204](#_Toc260755274)

[Rules 204](#_Toc260755275)

[Tasks 204](#_Toc260755276)

[Monitors 205](#_Toc260755277)

[Monitor Types 205](#_Toc260755278)

[Diagnostics 208](#_Toc260755279)

[Recoveries 208](#_Toc260755280)

[Cookdown 209](#_Toc260755281)

[Overview of Cookdown 209](#_Toc260755282)

[Multiple Workflows 209](#_Toc260755283)

[Multiple Workflows with Cookdown 210](#_Toc260755284)

[Criteria for Cookdown 212](#_Toc260755285)

[When to Configure for Cookdown 213](#_Toc260755286)

[Scripts Supporting Cookdown 213](#_Toc260755287)

[Multiple Property Bags 213](#_Toc260755288)

[Designing Custom Modules and Workflows 215](#_Toc260755289)

[When to Create a Custom Workflow 215](#_Toc260755290)

[Functionality Not Provided with Wizard 215](#_Toc260755291)

[Custom Modules 216](#_Toc260755292)

[Windows PowerShell 216](#_Toc260755293)

[When to Create Custom Modules 216](#_Toc260755294)

[Share Common Logic between Different Workflows 216](#_Toc260755295)

[Implement Complex Logic Not Possible with Existing Modules 217](#_Toc260755296)

[Improve Override Experience 217](#_Toc260755297)

[Custom Module Types 217](#_Toc260755298)

[Data Source Modules 217](#_Toc260755299)

[Probe Action Modules 218](#_Toc260755300)

[Probe Action Module Input Data and Triggers 218](#_Toc260755301)

[Condition Detection Modules 219](#_Toc260755302)

[Write Action Modules 219](#_Toc260755303)

[Passing data with parameters 219](#_Toc260755304)

[Values and Variables 220](#_Toc260755305)

[Explicit Values 220](#_Toc260755306)

[$Target Variables 220](#_Toc260755307)

[$Data Variables 220](#_Toc260755308)

[$Config Variables 221](#_Toc260755309)

[Overrideable Parameters 221](#_Toc260755310)

[Sample Workflow 222](#_Toc260755311)

[Creating Custom Modules and Workflows 229](#_Toc260755312)

[In This Section 229](#_Toc260755313)

[How to create a monitor based on a custom module 229](#_Toc260755314)

[How to create a monitor based on a Windows PowerShell script 236](#_Toc260755315)

[How to create a monitor and rule that share a script supporting cookdown 243](#_Toc260755316)

# Management Pack Authoring Guide

Welcome to the Management Pack Authoring Guide for Operations Manager 2007. This guide provides complete information on the design and implementation of management packs for System Center Operations Manager 2007.

## In This Section

[Management Pack Basics](#z67911cda31c94a2384b7a6f1e3d5bea3)

|  |
| --- |
| Covers the contents and structure of a management pack and provides an overview of editing tools. |

[Service Model](#ze4a377573ee94a3ab67e18b1e50b0191)

|  |
| --- |
| Covers concepts of classes, relationships, and discovery and provides guidance about how to design a service model for an application to be monitored. |

[Health Model](#zb43f85238807445d8f8ee26da359996d)

|  |
| --- |
| Defines different methods for measuring health state and collecting monitoring information for classes defined in the service model. Provides guidance about how to design a health model for an application to be monitored. |

[Composition](#z94c023257fa84110b5234f4b8b2f927c)

|  |
| --- |
| Defines the underlying the concepts and structure of monitoring workflows and provides guidance on creating custom workflows not available with standard Authoring Console wizards. |

[Presentation](#z6022bb3413c84414aeb641d85fab912d)

|  |
| --- |
| Provides guidance on defining and creating Views to display and analyze collected monitoring data in the Operations Console. |

# Management Pack Basics

The topics in this section provide an overview of the concepts required for building a management pack. These sections are not yet complete. When they are completed, they will provide detailed information on the topics described.

## In This Section

[What is a management pack?](#zc1efc510ddc84370bd37c0e536838819)

|  |
| --- |
| Provides an overview of the contents and structure of a management pack. |

[Editing Tools](#z87eac186338e4c818452d35d52083573)

|  |
| --- |
| Describes the different tools available for authoring a management pack and the capabilities of each. |

[Naming](#z63ee9cec607d482fa0cb0b05d51d57be)

|  |
| --- |
| Describes the naming requirements of management pack elements. |

[References](#z2a17c71d8cab45a19b0163e8ec4dbd4c)

|  |
| --- |
| Describes how management packs refer to elements in other management packs. Also provides details on sealing management packs to support external references. |

# What is a management pack?

All of the elements required for monitoring an application with Operations Manager 2007 including the service model, health model, views, and reports are stored in a management pack. A management pack is an XML file that conforms to a specific schema. It is installed into an Operations Manager 2007 management group and distributed to appropriate agents that use its contents to perform monitoring activities for a particular application.

When this section is completed, it will include details on the structure of a management pack and its contents.

# Editing Tools

There are multiple tools that are used to create and change management packs, depending on the requirements and skill level of the user. A single management pack might be changed using multiple different tools at different times and by different users.

The following are the different tools that are used to change management packs:

|  |  |
| --- | --- |
| Tool | Characteristics |
| Operations Console | **** Create or change a management pack in an existing management group. Changes are committed and deployed to agents immediately.  **** Create predefined monitoring scenarios by using templates and wizards. No ability to create monitoring scenarios outside predefined set.  **** Create distributed applications by using the Distributed Applications designer.  **** No ability to create custom classes or discoveries outside specific cases provided by templates.  **** Automatically generates names for management pack elements. No ability to implement standard naming scheme.  **** Primarily intended for users of Operations Manager 2007 with minimal knowledge of management pack authoring and basic customization requirements. |
| Authoring Console | **** Create or change a management pack on disk without requiring access to the management group. Provides the optional ability to load management packs from a management group and install them after modification.  **** Create predefined monitoring scenarios by using wizards. Ability to create custom monitoring scenarios outside predefined set. Requires some use of XML editor.  **** Create custom classes, relationships, and discoveries.  **** Ability to provide specific name to each management pack element.  **** Primarily intended for management pack authors with significant knowledge of management pack authoring. Allows for full customization but also exposes the author to more complexity. |
| XML Editor | **** Used by Authoring console for specific scenarios.  **** Can be used for scenarios not possible with the Authoring console such as copying existing management pack elements.  **** Can optionally be used for full management pack authoring.  **** Primarily intended for manually editing the management pack to augment features not provided by the Authoring console. Requires knowledge of XML. |

When this section is completed, it will include details on each of the tools and identify the scenarios under which each is most appropriate.

# Naming

All management pack elements have a name (or ID) and at least one display name. One element can have a different display name for each language included in the management pack. A single element though can have only one name, and this name must be unique for all management packs installed in a management group. The Operations console hides the name from the end user and automatically generates a unique name for any new management pack elements. The Authoring console exposes the name to the author who is required to provide a unique name for all elements in a management pack.

When this section is completed, it will include detail on the naming requirements in a management pack and recommendations on naming schemes to implement.

# References

Management packs rely on other management packs for elements such as base classes, monitor types, and modules. Library management packs are installed with Operations Manager 2007 to provide the core set of elements required by other management packs. In order for a management pack to use an element from another management pack, it must have a reference to that management pack. The reference defines an alias for the management pack containing the required element and the minimum version of the management pack that is required.

A management pack may only create references to sealed management packs. The sealing process uses a certificate to convert a management pack from XML to a binary file. In addition to supporting references, sealing a management pack allows versioning to be enforced.

When this section is completed it will provide details on creating references and using elements from other management packs. It will also provide instructions for sealing a management pack and further details on the implications of this action.

# Service Model

In Operations Manager 2007, all hardware, software, services, and other logical components of different applications that require monitoring are described in a service model. A model is a computer-consumable representation of software or hardware components that captures the nature of the components and the relationships between them. It can be thought of as a simplified representation of the application for monitoring.

When implementing monitoring for an application that uses Operations Manager 2007, the first step is to design a service model.

The topics in this section provide an overview of the key concepts that must be understood to perform this design, a basic process for designing a service model, and walkthroughs and examples of using the Operations Manager 2007 Authoring console to build the service model..

## In This Section

[Key Concepts](#zdfc9f4de2b66481785d40a2dab9666e5)

|  |
| --- |
| Explains the concepts of the different components in a service model. |

[Service Model Design Overview](#z879b344ff01b42d59d45634f474a2a56)

|  |
| --- |
| Explains the process and best practices when you design a service model for a particular application. |

[Building a Service Model](#zb3c71ae90f314d5381bf5bda8b85cbe8)

|  |
| --- |
| Demonstrates the different tasks used for building a service model for a sample application using the Authoring console. |

# Key Concepts

In Operations Manager 2007, all hardware, software, services, and other logical devices that require monitoring are described in a service model. A model is a computer-consumable representation of software or hardware devices that captures the nature of the components and the relationships between them.

## In This Section

[Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae)

|  |
| --- |
| Overview of classes and inheritance. |

[Relationships](#z332a9f5d49484136be5a6f3fc0b85218)

|  |
| --- |
| Overview of the different kinds of relationships between classes and their implications to monitoring. |

[Viewing Classes in the Consoles](#z995d1b41836743aebef04c569d8bfc1f)

|  |
| --- |
| Procedure that uses the Authoring console and Operations console to explain the concepts of classes, relationships, and inheritance. |

[Discovery](#zbe7560f04e134efd8884bcbb11d5bf3c)

|  |
| --- |
| Overview of the process for discovering instances of classes and relationships. |

[Discovery Scripts](#z874110d432734565a5ee6eaff0826a34)

|  |
| --- |
| Details about how to write a script to perform discovery. |

# Classes

The main element of a service model is a class. This represents some object being managed. It could represent a computer, a database, a service, a disk, an application, or any other kind of object that requires monitoring. A management pack may define any number of classes to represent the application it is monitoring in addition to using existing classes in other management packs and management pack libraries.

Each monitored object that appears in the Operations console is an instance of a particular class. All instances of a class have a common set of properties and a common means of being monitored. Each of these instances is created by a discovery that uses logic specific to the particular class to identify the instances and collect the values of their properties.

Note

Like all management pack elements, classes have an ID and a Display Name. These terms may be used differently in different consoles. In this documentation, ID will refer to the unique name of the class that is only seen in the Authoring console while Name and Display Name will refer to the language-specific name that appears in the Operations console.

## Properties

All instances of a particular class will share a common set of properties. The values for these properties are provided by discoveries and can vary among different instances. Properties are used to represent details of the object, such as a unique name, configuration settings, and other details that may be interesting to the user or that are required for monitoring scenarios.

### Key Properties

A key property uniquely identifies each instance of a particular class. If a property is marked as a key property, each instance of the class must have a unique value, and the value may not be null. For hosted classes, the value must only be unique for all instances of the class that have the same hosting parent. For unhosted classes, it must be unique for all instances of the class in the management group. Hosting relationships are discussed more in the [Relationships](#z332a9f5d49484136be5a6f3fc0b85218) section.

Classes do not always require a key property. A key property is only required if more than one instance of a class is expected for a single parent. If only a single instance is expected, a key property is not required but may still be defined.

For example, SQL Database Engine has a key property of Instance Name because a single computer can have more than one instance of SQL Server installed. When there are multiple instances on a particular computer, each instance must have a different value for Instance Name in order to clearly distinguish between the different objects. The IIS Web Server class, by contrast, does not define a key property because there can be only one instance installed on any computer.

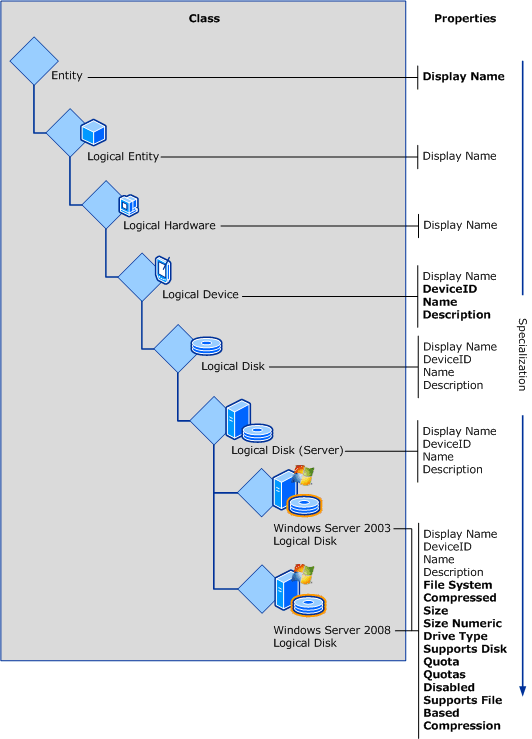
All objects have a Path Name property that is calculated from the object’s key property or properties and those of its hosting parent(s). For unhosted objects, the Path Name will be the key property of the class itself. The Path Name can be used to uniquely identify any instance of a class in the management group.

## Base Classes and Inheritance

Every class must specify a base class that identifies an existing class that the new one will specialize. The management pack libraries that are included with Operations Manager 2007 contain several classes that can be used as the base for custom classes in management packs. A management pack will typically have at least one class inheriting from a library class and potentially other classes inheriting from classes in the same management pack.

The concept of a base class can be illustrated with the Windows Server Operating System management pack. This management pack includes classes representing logical disks installed on the agent computer. The following diagram shows the classes Windows Server 2003 Logical Disk and Windows Server 2008 Logical Disk. These classes are both based on Logical Disk (Server) that is defined in the Microsoft.Windows.Server.Library mp file. Logical Disk (Server) is in turn based on Logical Disk, which itself is based on Logical Device, and so on through Logical Hardware, Logical Entity, and finally Entity. All classes can trace a similar inheritance path and will always end up at Entity, which is the root of the class structure in Operations Manager. This is the only class that does not have a base class, and all other classes eventually inherit from it.

Inheritance of properties between classes



Entity has a single property named Display Name. This is inherited by all classes inheriting from it. All classes eventually inherit from Entity. That is why all classes have a Display Name property. No other classes in this example have properties until Logical Device, which defines Name, Description, and DeviceID. DeviceID is specified as the key property. These properties are all inherited by Logical Disk and Logical Disk (Server). Logical Disk (Server) then adds the additional properties Size, Drive Type, and File System. The bottom-level classes that are specific to the version of operating system inherit the entire set of properties provided by those classes above them in the inheritance tree. This concept is illustrated in [Viewing Classes in the Consoles](#z995d1b41836743aebef04c569d8bfc1f).

Note

System.Library cannot be opened in the Authoring console because of the Entity class. The Authoring console performs checks to make sure that a management pack is valid before it opens it. One of these checks is to make sure that all classes have a base class, and because Entity does not have one, the Authoring console considers it an invalid management pack.

In addition to inheriting its properties and relationships, any monitoring targeted at a class will be inherited by any other classes that use it as a base. Additional monitoring and discovery can be added to the new class, and any management pack objects targeted at the new class will apply only to the new class and not the base class.

## Class Types

Most classes have one or more instances identified and created through the discovery process. These are also known as concrete classes because they actually have instances. Abstract classes and singleton classes are special kinds of classes that behave differently and are used for particular scenarios.

### Abstract Classes

Abstract classes have no instances and exist only to act as a base class for other classes. All properties and relationships that are defined at the abstract class level are inherited by child classes and do not have to be defined again. Any monitoring objects targeted at the abstract class are similarly inherited by lower-level classes. Most of the classes that are defined in management pack libraries are abstract, since they are only provided to act as base classes for classes that are defined in custom management packs.

Abstract classes are used where there is a common set of properties, relationships, monitoring, or grouping that can be defined across all further specializations of a class. In the previous example, all of the classes shown above Windows Server 2003 Logical Disk and Windows Server 2008 Logical Disk are abstract. They exist only for the lower-level classes to inherit from.

### Singleton Classes

Singleton classes are used when there is one and only one instance of a class. The class is the instance and always exists. No discoveries can be defined for singleton classes. Their single instance is instead being created when the management pack is installed. Similarly, a key property is not required for a singleton class, since it will only ever have a single instance. Common uses of singleton classes are for Groups and Distributed Applications, because there is only a single instance of these classes required throughout the management group.

## See Also

[Viewing Classes in the Consoles](#z995d1b41836743aebef04c569d8bfc1f)

[Defining Classes and Relationships](#z039ae803443b49a5a8281b2197e915c8)

[Creating Classes and Relationships](#ze2ee647ce5bb41dca88a5cc693cf602c)

# Relationships

Relationships are defined between classes to indicate an association between a particular instance of one class and the particular instance of another. There are three types of relationship as listed below and detailed in the following sections.

 Hosting relationship

 Containment relationship

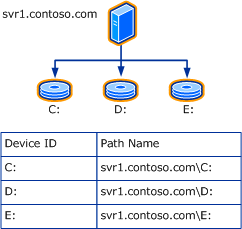
 Reference relationship

## Hosting Relationship Type

The most important and most restrictive relationship between classes is the hosting relationship. A class hosted by another class is called a hosted class, and an instance of the class is called a hosted object. If a class is not hosted by another, it is called an unhosted class, and an instance of the class is called an unhosted object.

When one object is hosted by another, that object relies on its hosting parent for its very existence. If the hosting parent is removed, the hosted child will also be removed. For example, a logical disk cannot exist without the computer that it is installed on. Therefore, the Logical Disk class is hosted by the Windows Computer class. This means that every instance of the Logical Disk class must have an instance of the Windows Computer class to host it. If a computer is removed from the management group, any instances of the Logical Disk class hosted by it will also be removed.

Sample Logical Disk hosting relationships



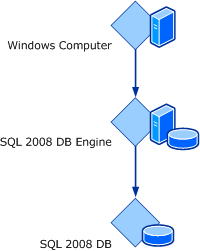
A hosted object can have only one hosting parent, but one parent can host multiple children. For example, a particular disk can be installed on only a single computer, but one computer can have several disks installed.

Hosted objects are affected by their hosting parents in the following ways:

|  |  |
| --- | --- |
| Identity | As described in [Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae), an object is identified by its key property. If the object is unhosted, the value of its key property must be unique for the whole management group. For hosted classes however, the key property value must be unique only for all objects that have the same hosting parent. To uniquely identify a hosted object, the key property of both the object and the object’s parent are required.  For example, the key property of the Logical Disk class is Device ID. Every computer in a particular environment will have a logical disk that has a Device ID of C: and possibly several other common disk letters. Because every disk object requires a unique value for its key property, the Logical Disk class cannot be unhosted. It is hosted by the Windows Computer class that has its own key property of Principal Name, which is the computer’s name. The key property only has to be unique for the hosting parent. In this case, this means that the Device ID only has to be unique for each computer. This makes logical sense, because a particular computer cannot have two disks that have the same Device ID.  The path for each Logical Disk instance will be the computer name followed by the Device ID, as shown in the previous diagram. |
| Managing Agent | If an object is hosted, it will be managed on the same computer as its hosting parent. If it has an instance of the Windows Computer class as a hosting parent (meaning that the object’s class is either hosted directly by the Windows Computer class or has the Windows Computer class somewhere in its tree of hosting parents), the object will be managed by the health service on that computer.  For example, the Logical Disk class is hosted by the Windows Computer class. This means that each Logical Disk object will be managed on the computer where the disk is installed.  If an object is unhosted, by default it is managed by the agent on the Root Management Server.  For example, groups are unhosted classes, and any monitors or rules targeted at a group are run on the RMS. |
| Available Properties | Any workflows targeted at a class have access to that class’s properties in addition to the properties of any of its hosting parent(s).  For example, a script in a monitor using the Logical Disk class as its target might require the name of the computer. Because an object can have only one hosting parent, we know the computer that hosts any particular instance of the Logical Disk class. The monitor can access the properties of the targeted object and the properties of that target’s hosting parent. |

The SQL Server management pack provides another example of hosting relationships. The hosting relationship between the Windows Computer class, the SQL 2008 DB Engine class, and the SQL 2008 DB class is shown here.

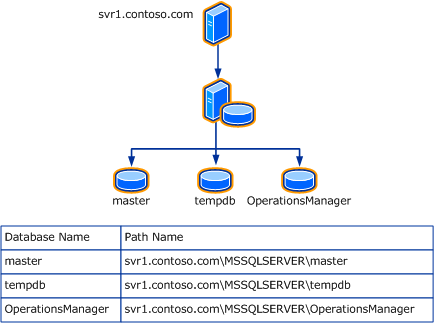
Hosting Relationships for SQL Server 2008 Classes



The SQL 2008 DB Engine class represents an instance of SQL Server 2008 installed on a particular computer. Because a database can be installed on only a single database engine, the SQL 2008 DB Engine class hosts the SQL 2008 DB class. There can be several databases with the same name in a management group, but any databases installed on a particular instance of the SQL Server class must have a unique name. The database engine, in turn, is hosted by the Windows Computer class. There can be several SQL Server instances with the same name in a management group. Each one on a particular computer must have a unique name.

Because there are two hosting relationships, the path name for each database will be the computer name followed by the instance name followed by the database name. An example is shown in the following diagram.

Sample Database hosting relationships



## Containment Relationship Type

The containment relationship type is less restrictive than the hosting relationship. It declares that one class is related to another class, although one is not required for the other.

Containment relationships are typically used for one of two purposes:

|  |  |
| --- | --- |
| Health rollup | Dependency monitors mean that the health of one object depends on the health of another object and can be associated with either a hosting or a containment relationship. If one object depends on the health of another, but the two are not in a common hosting relationship, a containment relationship between the two should be created. |
| Group membership | Objects are included in a group through a containment relationship between the group and the member object. |

Unlike a hosting relationship, a containment relationship is many-to-many. This means that one object can contain multiple objects, and a single object can be contained by multiple other objects. For example, one group can contain multiple objects, and a single object can be a member of multiple groups.

## Reference Relationship Type

The reference relationship is the most general of the available relationship types. A reference relationship is used when the parent and child classes are not dependent on one another; for example, a database could reference another database that it is replicating. One database is not dependent on the other, and the objects are discovered separately. Health rollup cannot be performed across reference relationships.

## See Also

[Creating Classes and Relationships](#ze2ee647ce5bb41dca88a5cc693cf602c)

# Viewing Classes in the Consoles

The following procedures illustrate the concepts of base classes and inheritance by using the Logical Disk classes in the Windows Server Operating System Management Pack. The first procedure inspects the definition of classes and relationships by using the Authoring console. The second uses the Operations console to view the same classes in actual operation.

Note

The term class is not shown in the Operations console but is instead known as a target. When you are working in the Authoring console, the term class is used. Each term in its relative context refers to the same concept.

## Prerequisites

This procedure assumes that you have the System Center Operations Manager 2007 R2 Authoring console installed and that you have access to a System Center Operations Manager 2007 R2 environment. The environment must have the Windows Server Operating System management pack installed. The procedure uses Windows Server 2008. However, this can be easily replaced with Windows Server 2003 if that is the only operating system version available in the environment.

To view a class definition in the Authoring console

|  |
| --- |
| 1. Start the Authoring console.  2. Open the file Microsoft.Windows.Server.2008.Discovery.mp.  3. In the Navigation pane, select Service Model, and then select Classes.  4. In the Classes pane, right-click Microsoft.Windows.Server.2008.LogicalDisk, and select Properties.  5. On the General tab, note in the Base Class text box that base class of the selected class is Microsoft.Windows.Server.LogicalDisk. This matches the diagram shown in [Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae). You could open Microsoft.Windows.Server.Library.mp that holds that class, and verify that its base class is System.Database, as shown in the diagram.  6. Click the Properties tab. Note the multiple properties on the current class that were inherited from Microsoft.Windows.Server.LogicalDisk, Microsoft.Windows.Server.LogicalDevice, and System.Entity. This shows the inheritance of properties by a class from its base class.  7. Click the Relationships tab. Note the relationships that the class is involved in. Of special note is the hosting relationship from Microsoft.Windows.ComputerHostsLogicalDevice in the Target section. This is the hosting relationship that the class inherits from its base class and matches the diagram in [Relationships](#z332a9f5d49484136be5a6f3fc0b85218). |

To view a class in the Operations console

|  |
| --- |
| 1. Start the Operations console.  2. In the Navigation pane, select Monitoring, and then select Discovered Inventory.  3. In the Actions pane, select Change Target Type. In the Select Items to Target dialog box, select View all targets.  Be aware that this list, which consists of all the classes included in all the management packs currently installed in the management group. Any of these classes may be selected to view a list of all its discovered instances and their properties. Any new classes included in a management pack that is installed later in the management group will be included in this list.  4. Select Windows Server 2008 Logical Disk, and then click OK.  Notice that this is the same class as inspected in the previous procedure in the Authoring console. Windows Server 2008 Logical Disk is the Display Name that is used in the Operations console, whereas Microsoft.Windows.Server.2008.LogicalDisk is the Name or ID of the class that must be unique. The view shows a listing of all instances of logical disk on Windows Server 2008 computers that were discovered in the current environment.  5. Select one of the instances.  Take note of the properties in the Detail View pane. These are the same properties that are seen in the previous procedure viewing the definition of the class. The current view shows the values for each property that were collected by the discovery process. Notice also the Path name property that is built from the key property of the current class and its parent(s). In this case, the key properties include the computer name and the device name.  6. In the Actions pane, again select Change Target Type.  7. In the Select Items to Target dialog box, select View all targets.  8. Select Logical Disk (Server), and then click OK.  This is the Microsoft.Windows.Server.LogicalDisk class that is the base class for Microsoft.Windows.Server.2008.LogicalDisk. The view resembles the previous one, but will include objects from both Windows Server 2003 and Windows Server 2008 (assuming both are installed in the environment). The properties are identical to the previous view, because the Windows Server Logical Disk class has the same properties that are inherited by Windows Server 2008 Logical Disk.  9. In the Actions pane, again select Change Target Type. In the Select Items to Target dialog box, select View all targets.  10. Select Logical Disk, and then click OK.  This is the Microsoft.Windows.LogicalDisk class that is the base class for Microsoft.Windows.Server.LogicalDisk. The instances are identical to the previous view, but fewer properties are shown. This is because the Microsoft.Windows.Server.LogicalDisk class has only the properties directly assigned to it, and inherits only its single property from System.Entity. The other properties are not visible because they are associated with a class further down the tree. |

## See Also

[Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae)

[Relationships](#z332a9f5d49484136be5a6f3fc0b85218)

# Discovery

The discovery process lets you locate instances of classes and relationships that are defined in a management pack and create them in the Operations Manager database. A discovery is an element that runs on agents creating new objects as they appear in the environment, changing the properties on existing objects, and removing objects as applications and components are uninstalled. Each class that is defined in the service model must have at least one discovery associated with it or no instances of the class will ever be created.

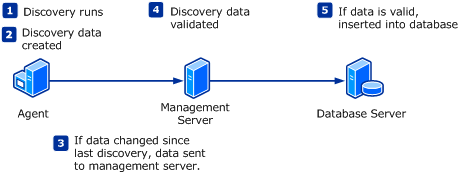
Note

Discovery does not apply to abstract or singleton classes.

## The Mechanics of Discovery

Discoveries collect information from the local computer to determine whether one or more instances of a particular class are installed and collect values for its properties. The discovery itself is only responsible for collecting this information. It sends the collected discovery data to the agent’s management server that is responsible for processing the collected data to insert into the Operations Manager database.

Discovery process



Discoveries are typically run on a scheduled basis. They continue to run on the agent even after instances are first discovered to determine whether changes were made. This interval will vary for different discoveries, balancing the considerations of the time until a new instance is discovered with the overhead generated from the discovery process.

The first time that a discovery runs on an agent, it sends its information to the agent’s management server. It also stores a copy of the collected data in the local cache. Each successive time that discovery runs, the data is only sent to the management server if a change is detected. Such changes could include a new instance, at least one property change to a previously discovered instance, or a previously discovered instance being removed. If any of these conditions are met, the discovery data is sent to the management server where it is processed.

When the management server receives the discovery data, it will inspect it to determine whether the data is valid. This includes checks such as verifying that all specified classes and properties are valid and that all required key properties were provided. If the discovery data is invalid, an error is generated on the management server and no discovery data is submitted. If the discovery data includes multiple instances and there is a problem with even a single instance, no data for any of the instances is submitted.

If the discovery data is valid, one of the following scenarios occurs.

 If any of the instances have not previously been discovered, they are then created.

 If any of the instances have previously been discovered, any properties that may have changed are replaced with the value in the discovery data.

 Any instances that were previously discovered but that are not in the current discovery data are removed.

Discovery data always includes the ID of the discovery itself and the ID of the target object. These values are then stored with all discovered objects. This is how data from a discovery can be compared to previously discovered data.

## Discovery Data

The primary responsibility of a discovery is to create discovery data. This includes the details of each class and relationship instance being discovered. A single discovery may only output a single set of discovery data. However, a single set of discovery data can include a single instance of a single class or relationship, multiple instances of the same class or relationship, or multiple instances of multiple classes and relationships.

Discovery data must include the following information.

 GUID of the discovery element itself

 GUID of the target object that the discovery is running against

 GUID of each class or relationship being discovered

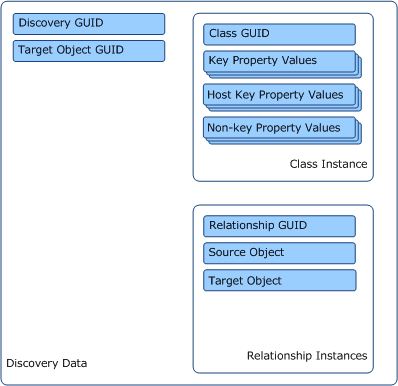
 Value for each key property of any class instances

 Value for each key property of any hosting parents of any class instances

 Source and target object for any relationship instances

In addition to this information, the discovery data can also include values for any non-key properties.

Conceptual view of discovery data



Each kind of discovery accesses this information through different means by using a combination of information that is provided by the management pack author and variables that are resolved when the management pack is run. Script discoveries in particular must be aware of each of these elements and explicitly provide a value for each.

## Discovery Data Sources

A discovery can access any information on the local agent to collect required information. The source of information will vary for different applications and for different classes and relationships. Because a management pack can include custom modules able to access almost any available data, there is almost no limit to the sources that a particular discovery may use. There are a common set of data sources that apply to most scenarios however, and these are supported by modules included in management pack libraries and wizards in the Authoring console.

### Registry

Accessing the registry on the local computer is a preferred method of discovery because of the small overhead required. The registry frequently contains lots of easy to access configuration information for many applications. A discovery can check whether a particular key or value exists, or it can collect the data from any value. Collected values may be either only collected for a property or compared to some known value to determine whether discovery data should be created.

For example, a discovery may have to check for a particular registry key to indicate that its application is installed on the local computer. If only a particular version of that application is supported by the management pack, a registry value holding the application version may be inspected to determine whether an instance should be created. Other values in the registry may then be collected to populate properties of the discovered instance.

### WMI

WMI can provide lots of information about a computer or Windows operating system components. It can also be the source of information for any application that includes a WMI provider exposing parts of the application through a WMI interface. A discovery can run a WMI query to determine whether instances of a particular WMI class exist and to collect the properties from any returned instances.

For example, a particular management pack might collect information from the computer BIOS, that can be obtained by using the WMI32\_BIOS WMI class.

### Scripts

If a discovery requires data inaccessible from the registry or WMI, or if the discovery needs more complex logic than those discoveries provide, a discovery script must be used. A discovery script can be written in any language that can access the MOM.ScriptAPI COM object that is installed with the Operations Manager agent. The most typical scripts are written either in a language supported by Windows Script Host (VBScript or Jscript by default) or Windows PowerShell.

Discovery scripts that use VBScript or Jscript can be created by using modules in management pack libraries and wizards in the Authoring console. Modules are available for discovery scripts that are written in Windows PowerShell in Operations Manager R2 but are not supported by wizards in the Authoring console. A custom discovery would have to be created by using the appropriate module.

Any management pack requiring support by Operations Manager 2007 SP1 cannot use the Windows PowerShell modules because they are only installed with the Operations Manager R2 agent. In this case, custom discoveries that use Windows PowerShell can be created by using a module that runs a command, configuring it to build the powershell.exe command line and starting the discovery script.

## Discovery Targets

The target of a discovery determines where the discovery will run. Any agent managing an instance of the target class will run the discovery. In order to minimize overhead, the target for a discovery should represent the most specific class where the class being discovered may potentially be found.

For example, a discovery determining whether an application is installed on a computer may have to target a broad class, such as the Windows Operating System class, because the application could be potentially installed on any computer in the environment. Another discovery responsible for a particular component of that application could target the newly discovered application class, because the component would presumably never be discovered on a computer where the application wasn’t installed.

## Discovering Relationships

In addition to instances of classes, instances of relationships must also be discovered. Hosting relationships are discovered automatically as part of the discovery of hosted objects, but containment relationships must be explicitly discovered.

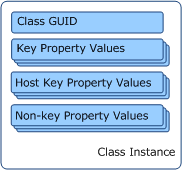
### Hosting Relationships

Because a hosted object cannot exist without its hosting parent, the parent object must either already have been created before the child can be discovered or must be discovered at the same time as the child. Discovering the objects at the same time means that both classes are discovered by the same discovery with instances included in the same set of discovery data.

Instances of hosting relationships are not explicitly discovered but are created automatically when a hosted object is discovered. If the object is hosted, the discovery data must provide the key property of any of its hosting parents in addition to the key property for any instances that it includes. This is sufficient information to identify the source and target objects of the hosting relationship so that it can be automatically created.

For example, a class may be based on the Windows Computer Role class representing a particular application installed on a computer. Because the Windows Computer Role class is hosted by the Windows Computer class, the application’s class is also hosted by the Windows Computer class. The discovery for the class must provide the computer name because this is the key property of the Windows Computer class, that is, the hosting parent of the object being discovered. If the class also had its own key property, this would have to be included. If it did not have a key property, the computer name would be the only key property that would have to be provided.

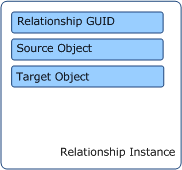
Conceptual view of class instance



### Containment Relationships

Unlike hosting relationships, containment relationships must be explicitly discovered. The source and target objects for the relationship instance must either already have been discovered or be included in the same discovery. Containment relationships between specific objects are usually discovered with a discovery script. The resulting discovery data specifies the GUID of the relationship in addition to the source object and target object for the relationship instance that should be created.

Conceptual view of relationship instance



### Populating Groups

Groups are populated by creating instances of the containment relationship between the group objects and its members. You can do this by using a discovery script like other containment relationships, or a special group population module can be used. Group population is different from other kinds of discoveries because it is performed on the Root Management Server by using information from the Operations Manager database instead of running on the agent that uses data that is collected from the local computer. This process is typically easier and more efficient than manually creating the containment relationship instances.

The Authoring console does not provide a wizard for implementing group population. This kind of discovery must be created manually by using a special Group Populator module.

## Proxy Discoveries

Most discoveries discover objects managed by the local agent computer. There are scenarios where an agent may have to perform discovery of a particular object or set of objects that are either unhosted or hosted by another computer. An example of this scenario may include collecting information from a Configuration Management Database. In this case, one agent might be responsible for running a discovery that uses a script to perform queries against information that is stored in the database. This information is then used to create instances and set property values for objects that are located on other computers. The agent running the discovery is acting as a proxy for the other agents.

The proxy scenario is typically performed with a script because other kinds of discovery are typically unable to access the required data or perform the required logic. As previously discussed, discovery data requires the key property of each discovered object and any of their hosting parents. In the proxy scenario, the hosting computer will differ from the agent running the discovery, and the discovery merely needs to provide this computer name in the discovery data. The object will be discovered and managed by the specified agent.

In order to perform discovery on behalf of another agent or discovery of an unhosted object, the agent running the discovery must have its proxy setting enabled. If this proxy setting is not enabled, the discovery will generate an error, and the discovered instances will not be created. The proxy setting is configured in the Agent Properties, which are accessed from the Administration pane in the Operations console.

## Discovery on Demand

Most discoveries are performed on a scheduled basis. New objects or changes to existing objects are not discovered until the next time that the discovery for the particular class is scheduled to run. Discoveries with intervals that are too frequent put too great a load on the agent performing discovery, whereas discoveries that are too infrequent can take too much time to recognize such changes.

Discoveries can be designed to be executed in response to a particular event, assuming that an application generates such an event in response to a configuration change. In this case, the discovery could be run immediately after the configuration change and may not have to be scheduled. A discovery could also be designed to be run on demand by a user through an Agent Task.

The Authoring console provides wizards for scheduled discoveries only. Discoveries that run on demand or in response to an event may be created by using the Authoring console, but they must be created with custom discoveries that use custom modules.

## See Also

[Defining Discoveries](#ze8857ff93cf24bcf90913ae6ce2913b5)

[Creating Discoveries](#z6014eb924a6d440aa079a22814740bd5)

# Discovery Scripts

Discovery scripts are used when complex logic is required for discovery of a particular class or relationship, or when required information cannot be accessed from the registry or WMI. The script collects data from information on the agent and creates discovery data by using the MOM.ScriptAPI object that is installed with the Operations Manager 2007 agent.

## Creating Discovery Scripts

Discovery scripts may be written in any script language that can access the MOM.ScriptAPI object. The most common languages used are VBScript, JScript, and Windows PowerShell. Discovery scripts that use VBScript or Jscript can be created by using modules in management pack libraries and wizards in the Authoring console. Modules are available for discovery scripts that are written in Windows PowerShell in Operations Manager R2 but are not supported by wizards in the Authoring console. A custom discovery would have to be created using the appropriate module.

Management packs that require support by Operations Manager 2007 SP1 cannot use the Windows PowerShell modules, because they are only installed with the Operations Manager R2 agent. Discovery scripts that use Windows PowerShell can be created by using modules that run a command and configure them to build the powershell.exe command line.

## Required Arguments

Discovery data must include the GUID of the target object that the discovery is running against and the GUID of the discovery itself. The value for each is required when the script calls the CreateDiscoveryData method. They will be unknown when the script is written. However, they can be accessed through variables and sent into the script through arguments. The GUID of the discovery is accessed by the $MPElement$ variable, whereas the GUID of the target object is accessed with $Target/Id$.

Other arguments can be sent into the script as required for the particular discovery.

## MPElement Variables

Discovery data specifies the classes and relationships being discovered in addition to their properties. The data must include the GUID for these values instead of simply their names. The GUID of any management pack element can be retrieved by using an $MPElement variable, and these are used frequently in discovery scripts.

The syntax of an $MPElement variable is as follows:

$MPElement[Name="ElementName"]/SubElementName$

Example uses of the variable are as follows:

|  |  |
| --- | --- |
| Class in the same Management Pack file. | $MPElement[Name="MyMP.MyClass"]$ |
| Class in a different Management Pack file. | $MPElement[Name="Windows!Microsoft.Windows.Computer"]$ |
| Class property in the same Management Pack file. | $MPElement[Name="MyMP.MyClass"]/MyProperty$ |
| Class property in a different Management Pack file. | $MPElement[Name="Windows!Microsoft.Windows.Computer"]/PrincipalName$ |

## Basic Script Structure

The following sample shows a discovery script with the following characteristics.

 The script discovers a single instance of a class named MyMP.MyClass.

 The class is hosted by Windows Computer.

 The class has a key property named Name.

 The class has a non-key property named Version.

SourceId = WScript.Arguments(0)

ManagedEntityId = WScript.Arguments(1)

sComputerName = WScript.Arguments(2)

Set oAPI = CreateObject("MOM.ScriptAPI")

Set oDiscoveryData = oAPI.CreateDiscoveryData(0, SourceId, ManagedEntityId)

Set oInstance = oDiscoveryData.CreateClassInstance("$MPElement[Name='MyMP.MyClass']$")

oInstance.AddProperty "$MPElement[Name='Windows!Microsoft.Windows.Computer']/PrincipalName$", sComputerName

oInstance.AddProperty "$MPElement[Name='MyMP.MyClass']/Name$", "svr01.contoso.com"

oInstance.AddProperty "$MPElement[Name='MyMP.MyClass']/Version$", "1.0"

oDiscoveryData.AddInstance(oInstance)

oAPI.Return(oDiscoveryData)

## Script Breakdown

Details of each section of the script are discussed here.

SourceId = WScript.Arguments(0)

ManagedEntityId = WScript.Arguments(1)

sComputerName = WScript.Arguments(2)

The first two lines of the script accept the GUID for the discovery and the target object. These lines should stay unchanged in most discovery scripts, but they will typically be followed by other variables accepting arguments specific to the particular script. The next line gets a value for the computer name that is used for the key property of the hosting Windows Computer object.

Set oAPI = CreateObject("MOM.ScriptAPI")

Set oDiscoveryData = oAPI.CreateDiscoveryData(0, SourceId, ManagedEntityId)

The next two lines create a discovery data object by using the provided GUIDs. These lines will also be unchanged in most discovery scripts. The main purpose of the rest of the script will be to add detail to the discovery data object by using data collected from the agent computer.

Set oInstance = oDiscoveryData.CreateClassInstance("$MPElement[Name='MyMP.MyClass']$")

After the discovery data object is created, it can be used to create a new class instance. This requires an $MPElement variable that uses the name of the class. This variable translates to the appropriate GUID representing the class. This line would presumably be preceded by code that would collect data from the agent computer to determine whether one or more instances should be created in addition to populating variables for the instances properties that are populated with the following lines.

oInstance.AddProperty "$MPElement[Name='Windows!Microsoft.Windows.Computer']/PrincipalName$", sComputerName

oInstance.AddProperty "$MPElement[Name='MyMP.MyClass']/Name$", sName

oInstance.AddProperty "$MPElement[Name='MyMP.MyClass']/Version$", sVersion

With the instance now created, its properties can be populated. Any key properties of the class being discovered and the key properties of any of its parents must be provided. Values for other properties are optional. In this example, the class being discovered is hosted by Windows Computer and the key property of that class is added to the instance.

oDiscoveryData.AddInstance(oInstance)

After the instance is created and its properties populated, the instance is added to the discovery data object. If the script created multiple instances, a loop could be used that added each instance as it was created and configured. If this line is left out of the script, the instance is never added to the discovery data object.

oAPI.Return(oDiscoveryData)

After all instances are created and configured, the discovery data is returned into the workflow. This line is required, and without it the discovery data is discarded when the script ends. A discovery script may output only a single set of discovery data. If multiple instances are being discovered, the line appears only one time at the end of the script after all instances are added.

## Windows PowerShell

The following script is the Windows PowerShell equivalent of the VBScript sample discovery script shown previously. Both scripts use the same MOM.ScriptAPI object and methods.

param($sourceId,$managedEntityId$computerName)

$api = New-Object -comObject 'MOM.ScriptAPI'

$discoveryData = $api.CreateDiscoveryData(0, $sourceId, $managedEntityId)

$instance = $discoveryData.CreateClassInstance("$MPElement[Name='Demo.StoreApp.CentralQueue']$")

$instance.AddProperty("$MPElement[Name='Windows!Microsoft.Windows.Computer']/PrincipalName$", $computerName)

$instance.AddProperty("$MPElement[Name='MyMP.MyClass']/Name$", "svr1.contoso.com")

$instance.AddProperty("$MPElement[Name='MyMP.MyClass']/Version$", "1.0")

$discoveryData.AddInstance($instance)

$discoveryData

Other than obvious syntax, there are two primary differences between the VBScript and Windows PowerShell scripts, as follows.

param($sourceId,$managedEntityId)

The first difference is in the means of retrieving script arguments. Windows PowerShell can accept positional arguments as VBScript does, but named parameters that use the param command better support the Windows PowerShell modules included with Operations Manager 2007 R2. These modules let parameters be separately specified in the module configuration where they are retrieved by the script as named parameters.

$discoveryData

Another important difference is the way that the discovery data is returned to the workflow. In VBScript, the Return method of the MOM.ScriptAPI object is used. In Windows PowerShell , this method will not return the data correctly. The Return method sends the discovery data to the Standard Out (StdOut) stream, whereas Windows PowerShell requires the data to be sent to the output pipeline. This is achieved by typing the discovery data variable on its own line.

## Empty Discovery Data

If a discovery script does not return discovery data, an error will be generated on the agent. Even if no instances of the class are found, empty discovery data should be created and returned by the script. If no data is returned, Operations Manager cannot determine if any instances that were previously discovered have now been removed.

## See Also

[How to Create a Script Discovery](#z9edec4621dbe41e58335828beeee6e98)

# Service Model Design Overview

Designing a service model for an application consists primarily of defining a set of classes to represent the different application components and relationships between those elements to support the desired monitoring functionality. Simple applications may be adequately monitored with a model as basic as a single class. An application that consists of little more than a single Windows service, for example, may have only one class that is discovered on the appropriate computers and can act as the target for rules and monitors by using information generated by the application.

More complex applications will typically require multiple classes to represent different application components. Applications that may be installed across multiple computers may also benefit from classes used for consolidated health rollups for the application itself and for different application features composed of multiple components.

## In This Section

[Defining Classes and Relationships](#z039ae803443b49a5a8281b2197e915c8)

|  |
| --- |
| Basic process for defining a set of classes and relationships to represent the application. |

[Choosing a Base Class](#z6709d79030034a5680b92a3bd9815bd6)

|  |
| --- |
| Criteria and recommendations for selecting a base class for custom classes. |

[Defining Discoveries](#ze8857ff93cf24bcf90913ae6ce2913b5)

|  |
| --- |
| Basic process for defining discoveries for each class in a service model. |

[Practices to Avoid](#z15f142ab17264104af01d2cfe12eb1fb)

|  |
| --- |
| Common practices to avoid when designing a service model. |

# Defining Classes and Relationships

The basic strategy of selecting a set of classes to provide a model for an application is to identify the application’s different components and define an appropriate class to represent each one. Each class definition will include its base class, its relationships with other classes, and a set of properties to hold the values required for the different monitoring scenarios targeted at the class.

Because a service model can be created for any kind of application on various computing platforms or even to support monitoring of physical devices, it is difficult to provide a single process that applies to all scenarios. The following guidance applies to designing a model for a typical application that is running on one or more Windows-based servers. Many applications fit into this basic architecture and can have an effective model designed with this fairly simple process. This basic strategy can also be extrapolated to other kinds of applications.

Each class listed here includes a short description of the base class that is typically used. Additional detail on the implications of the base class and criteria for selecting an appropriate one are provided in [Choosing a Base Class](#z6709d79030034a5680b92a3bd9815bd6).

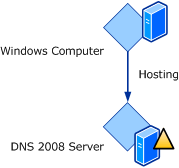
## Define Application Roles

The first class that should be defined is one to represent the installation of the application on a particular computer. This can be known as an application role. For many applications, a single class will be sufficient for this purpose. However, more complex applications may have multiple classes to represent different kinds of servers or different core services that require unique monitoring. This class is typically hosted by the Windows Computer class so that it can be discovered on any agent where the application is installed.

One thing to consider in this step is which application roles might be installed on separate computers. One function of a class is to act as a target for workflows specific to the application that has workflows targeted at the class running on the agent managing an instance of that class. If two components of an application will always be installed on the same computer, they can frequently be represented by a single class. If there is a potential for them to be installed separately however, they should each have their own class so that any targeted workflows will run on the appropriate agent.

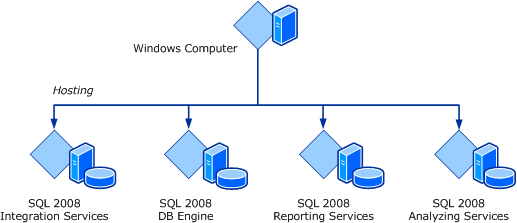
The DNS 2008 Server management pack provides an example of a single class used to represent an application. An instance of the DNS 2008 Server class is discovered on any computer that is running Windows 2008 Server that has DNS installed. Any monitors and rules specific to the DNS Server class can be targeted at this class in addition to discoveries for identifying other DNS components that may be installed on the server.

DNS 2008 Server class



The SQL Server 2008 management pack, on the other hand, has four separate classes for its primary functions: the DB Engine class, the Analysis Services class, the Integration Services class, and the Reporting Services class. These are distinct services in SQL Server that will each have unique monitoring and can be installed separately from another. SQL Server requires a separate class for each role in order to independently measure the health state of the different components and use them as targets for unique monitoring scenarios.

Computer Role classes for SQL Server 2008



The base class used for application roles will typically be the Windows Computer Role class or the Windows Local Application class. These are both hosted by the Windows Computer class. Classes based on the Windows Computer Role class are considered a primary function of a server, and their health state will be automatically rolled up to their hosting computer. The Windows Local Application class is intended to represent smaller applications, and its health state will not roll up to the computer.

Computer role classes typically do not have a key property, because they usually have only a single instance on any given computer. There are exceptions to this standard, however, for applications that can have multiple instances installed. SQL Server, for example, can have multiple instances installed on a single computer, and the class for each role has a key property for the instance name.

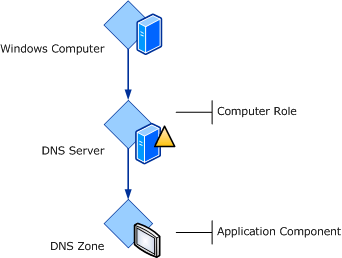
## Define Application Components for Each Application Role

After each of the application’s primary roles are identified (a single role for a simple application), identify those components hosted by each role that require a distinct health state. The application role alone may be sufficient for a simple application, but more complex applications may benefit from one or more additional classes for additional components of the application. These components will typically be hosted by one of the computer role classes that are defined in the previous step or by another application component.

Application components typically share a one-to-many relationship with their hosting class. In other words, a single computer role instance will usually hold multiple instances of a single application component. If only a single instance of the application component is expected, it is frequently of minimal value to create a unique class for it.

For example, the DNS Server class hosts a class named DNS Zone. Each instance of the DNS Zone class represents the different zones that can be installed on a single DNS server. Creating a separate class lets each zone have its own health state.

DNS Server classes



Another example is the Internet Information Server (IIS) 2008 management pack, which defines a class named IIS 2008 Server Role that is discovered on any computer that has IIS 2008 installed. This class hosts classes representing the different kinds of components that may be installed: Web server, FTP server, and SMTP server. Each IIS installation may have any combination of these installed. Each Web server, in turn, may have multiple Web sites that are represented by the IIS 2008 Web Site class. In this case, the server role hosts the Web server, and the Web server hosts the Web site. The same is true for FTP sites.

Application Component classes for IIS 2008



Application component classes should typically use the Windows Application Component class for a base class. This abstract class is unhosted. This means that the class inheriting from it can create its own hosting relationship with the application’s computer role. It will typically require a key property, because multiple instances of the class are usually expected for the hosting role.

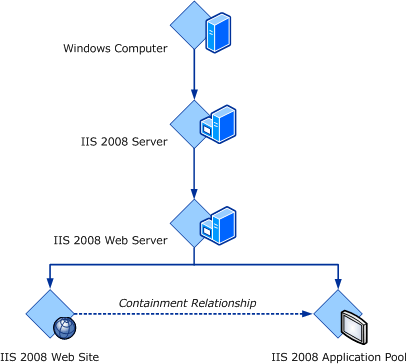
## Identify Dependencies Between Classes

The health state of any class can depend on the health state of another as long as they share a hosting or containment relationship. For example, when a monitor targeted at an application role class reports a negative health state, the computer that is hosting the object can also show a negative state. Similarly, the health of an application role can depend on any of the application components that it hosts.

Most of the health dependencies required for a service model will already exist by the time that classes and hosting relationships are defined. Containment relationships have to be defined for any additional classes whose health depends on the health of another class that it does not already share a hosting relationship with.

For example, in the IIS 2008 management pack, the IIS 2008 Web Server class hosts classes called IIS 2008 Web Site and IIS 2008 Application Pool. Web sites and application pools are both components that can be created independently on a Web server, and any particular instance of one can exist without any particular instance of the other. However, one or more Web sites can be configured to use a particular application pool, and the health of the Web site will depend on the health of any application pool that it is configured to use. In order to support this dependency, the IIS 2008 management pack includes a containment relationship between the IIS 2008 Web Site class and the IIS 2008 Application Pool class.

Containment relationship between IIS 2008 classes



## Define Application Class

Distributed applications typically define a top-level class to represent the application itself. Containment relationships are defined between this class and each of the application’s role classes (which in turn, host the application’s components). This provides an overall health state for the application. All applications will not have such a class. It is created only for those applications where a single consolidated health state for all the application’s roles and components in a particular environment is relevant.

For example, the Active Directory management pack includes a class named Active Directory Topology Root. This represents the overall Active Directory installation in a particular environment. This class contains classes for the Active Directory Forest, Sites, and Domains so that it can provide an overall health state for the environment.

The SQL Server management pack, by contrast, does not have a class representing the whole environment. The nature of SQL Server is such that there is no concept of an overall environment but instead distinct installations on different servers. There would be minimal value in providing an overall measurement for the health of all instances of SQL Server.

The application class should typically be a singleton. Because it represents the overall application installation in the environment, there should not be a requirement for multiple instances of it. If the class is based on the Distributed Application class, it will be included in the list of Distributed Applications in the Operations console.

## Groups

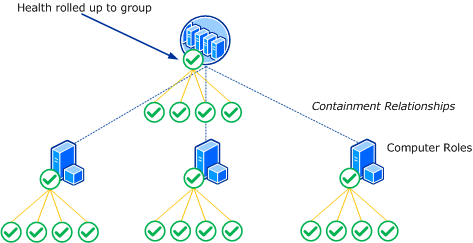
The final step in the service model design is to define any required groups. Groups are singleton classes that contain a collection of other objects. They can be useful in a service model for different reasons, as detailed here.

### Health Rollups

Groups can be used for a health rollup for instances of a particular class. For example, a distributed application may include multiple computers that have a particular computer role installed. In addition to a health state for each computer role instance and a health state for the whole application, a health state may be required for all instances of the particular role. This functionality is supported by creating a group that contains all instances of the class to consolidate.

An effective exercise of identifying any required health rollup groups is to draw the desired diagram view for the application. This should include all the classes created in the previous steps. If these classes are insufficient to create the required diagram view, health rollup groups can be defined to complete the requirements.

Rollup of health from classes to containing group



Groups used for health rollups can be based on either the Instance Group class or the Computer Role class. If based on the Instance Group, they are the same as any other instance group and will appear in the Groups list in the Authoring pane of the Operations console. If they are based on the Computer Role class, they will not appear in this list.

### Instance Groups

Instance groups can be used to distinguish a set of objects that do not have a unique class for monitoring. These are the same as typical instance groups that a user can create in the Operations console.

For example, the Active Directory 2008 management pack includes a group named RODC Group that contains all read-only domain controllers. These are instances of the Active Directory Domain Controller Server 2008 Computer Role class for which the Read-Only Domain Controller property is true. This group is then used for overrides that disable rules that are not relevant to a read-only domain controller. One alternative would be to create a special class for read-only domain controllers inheriting from the Domain Controller class. However, it was determined that there were not enough unique monitoring scenarios for the read-only domain controller to warrant specializing the Domain Controller class. The RODC Group is sufficient to achieve the desired functionality.

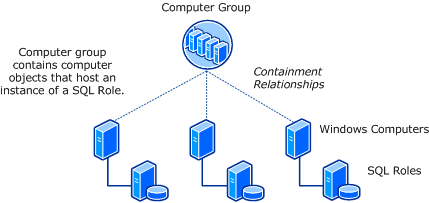
These groups are typically based on the Instance Group class. Any group based on this class will appear in the Groups list in the Authoring pane of the Operations console.

### Computer Groups

Computer groups contain a collection of computer objects on which another object exists. This is distinctly different from the instance groups containing the application objects themselves. This type of group can be used to support such group functionality as views and user roles for the computers hosting a particular application.

For example, the SQL Server management pack includes a group called SQL 2008 Computers that contains all computers with at least one SQL Server 2008 component installed. This can be used to provide a database administrator access to the servers hosting SQL Server through a user role or a view of all computers that are running instances of SQL Server in a state view.

Computer groups for SQL Server 2008



Computer groups are based on the Computer Group class. Any group based on this class will appear in the Groups list in the Authoring pane of the Operations console.

## See Also

[Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae)

[Relationships](#z332a9f5d49484136be5a6f3fc0b85218)

[Choosing a Base Class](#z6709d79030034a5680b92a3bd9815bd6)

[Creating Classes and Relationships](#ze2ee647ce5bb41dca88a5cc693cf602c)

# Choosing a Base Class

Each class that is defined in the service model must have a base class, and selecting the appropriate one is important to achieve the expected operation of the class in the overall service model. The base class may be selected from a class available in a management pack library or an abstract class that is defined in your management pack.

## Criteria for Selecting a Base Class

The following should be considered when you select a base class for a particular class. For more information, see the links here.

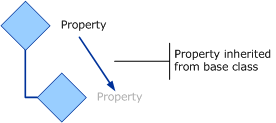
### Inheritance

A class will inherit various characteristics of its base class. These characteristics affect which base classes you can use from management pack libraries and which abstract classes that you might create to have other classes inherit from.

#### Properties

Typically, abstract classes in management pack libraries will not have properties other than the Display Name property inherited from the Entity class. Abstract classes that are defined by your own management pack may define properties that are inherited by other classes. This saves the lower level classes from each replicating the same property definition.

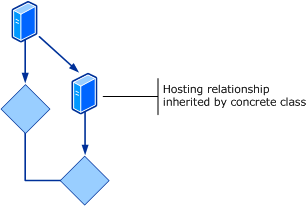
Conceptual view of property inheritance



#### Relationships

If a class is hosted, any class that uses it as a base class will be hosted by the same parent class. For example, the Windows Computer Role class is frequently used as a base class and is hosted by the Windows Computer class. Any classes that use the Windows Computer Role class as a base class will also be hosted by the Windows Computer class. If a class is to be hosted by a class other than the Windows Computer class, it must use another base class, such as the Windows Application Component class, which is unhosted. Because that base class is unhosted, the class will either remain unhosted or have a custom hosting relationship that is defined for it.

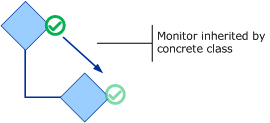
Conceptual view of relationship inheritance



#### Monitoring

A class will inherit any monitoring targeted at its base class. This is actually how the standard set of aggregate monitors (Availability, Performance, Security, and Configuration) is applied to all classes. These monitors are applied to the Entity class in the System library. Because all classes inherit from this class, all classes inherit those aggregate monitors.

Conceptual view of workflow inheritance



### Logical Grouping

In addition to inheritance, use of a custom abstract class in your management pack can help in logically grouping common classes. For example, you may have multiple kinds of computer roles in the application that will each inherit from the Windows Computer Role class. Even if there will be no common monitoring between the classes, there is still value in creating a single abstract class that will act as a base class for individual classes representing each role. This would enable such functionality as creating a single view that targets the abstract class and which would include all computer roles in the application. This also allows for easy specification of all computer roles for criteria in groups and overrides. It basically gives other management pack elements the option of directly targeting a single specific class or all the related classes, depending on the particular scenario.

### Side Effects

Different base classes in management pack libraries may have special functionality that is specific to that class. For example, any class that uses the Distributed Application class as a base class is considered a distributed application and will be included in that view in the Operations console. Some classes in the management pack libraries are intended for internal purposes only and should not be used by custom management packs because of unintended effects.

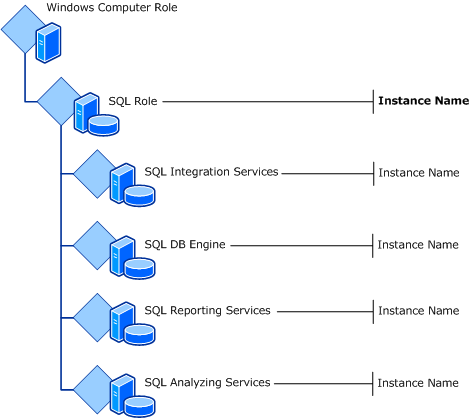
Instead of individually documenting all classes, you should limit your use of abstract classes from the management pack libraries to the list of common base classes here. Any side effects from the particular class are described in this table.

## Custom Base Classes

Rather than using only abstract classes from management pack libraries, it is a common strategy to create abstract classes within your management pack for other classes to inherit from. Such a class can provide the previously discussed benefits of inheritance and logical grouping for similar classes in your management pack.

For example, the SQL Server management pack includes four classes for each of its primary server roles. Each should inherit from the Windows Computer Role class and be hosted by the Windows Computer class. However, instead of directly inheriting from this class, an abstract class named SQL Role is created based on the Windows Computer Role class. The individual computer role classes are then based on this custom class.

SQL Server Computer Role Classes



With this strategy, the management pack can easily include views by using the SQL Role class as a target that displays information for all SQL-related services. Also, because each of the SQL services can have multiple instances installed on a single computer, they require a key property. Instead of including this property on each class, it is defined on the SQL Server Role class and is inherited by the other classes. Each class then adds additional properties specific to that service.

## See Also

[Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae)

[Relationships](#z332a9f5d49484136be5a6f3fc0b85218)

[Defining Classes and Relationships](#z039ae803443b49a5a8281b2197e915c8)

[Creating Classes and Relationships](#ze2ee647ce5bb41dca88a5cc693cf602c)

# Recommended Base Classes

The following table provides the list of classes from management pack libraries that should be used as the base class for classes that are defined in your management pack. They may be used directly as the base class for concrete classes or for custom abstract classes that other classes in your management pack will inherit from. This set of classes will be sufficient to achieve almost any scenario required by any service model.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class ID | Display Name | Library | Hosting Relationship | When to Use | Effects |
| Microsoft.Windows.ComputerRole | Windows Computer Role | Microsoft.Windows.Library | Hosted by Windows Computer. | Indicates a primary role of a server. | Health automatically rolled up to computer. |
| Microsoft.Windows.LocalApplication | Windows Local Application | Microsoft.Windows.Library | Hosted by Windows Computer. | Application that is running on a computer but not the primary server role. | Health not automatically rolled up to computer. |
| Microsoft.Windows.ApplicationComponent | Windows Application Component | Microsoft.Windows.Library | Unhosted | A component of an application. | A hosting relationship can be created by using another class, typically based on ComputerRole or LocalApplication. |
| System.Service | Distributed Application | System.Library | Unhosted | Represents an application for health rollup. | Instances listed in the Distributed Applications view. |
| System.ApplicationComponent | Application Component | System.Library | Unhosted | Used for application components not hosted by a computer that is running Windows. |  |
| Microsoft.SystemCenter.InstanceGroup | Instance Group | Microsoft.SystemCenter.InstanceGroup.Library | Unhosted  Singleton | Base class for groups of instances of one or more classes. | Included in the Groups list in the Operations console. |
| Microsoft.SystemCenter.ComputerGroup | Computer Group | Microsoft.SystemCenter.Library | Unhosted  Singleton | Base class for computer groups. | Included in the Groups list in the Operations console. |
| System.Group | Group | System.Library | Unhosted  Singleton | Base class for groups used for health rollup. | Not included in the Groups list in the Operations console. |

# Defining Discoveries

After a set of classes and relationships is defined for an application, a method of discovering each must be defined. No instances of a class will be available for monitoring in Operations Manager until they are discovered.

## Identifying Discoveries for Each Class

There are three basic questions that must be answered for each class requiring discovery.

 What source or sources hold the data required to determine whether an instance should be created and to populate its properties? This defines the kind of discovery.

 What agent hosts the required data source? This defines the target of the discovery.

 How frequently should the discovery be run? This defines the interval of the discovery.

## Basic Discovery Strategy

The discovery process for an application model composed of multiple classes will typically be performed in multiple steps, with each step including one or multiple discoveries. A basic set of classes is first discovered to identify the agents where the application is installed. These classes act as targets for successive discovery of other classes.

For the basic model described in [Defining Classes and Relationships](#z039ae803443b49a5a8281b2197e915c8), discovery can typically be performed in three basic steps.

1. Discover application role classes to identify those agents with the application installed. This is typically performed with registry discoveries targeting a broad class, such as Windows Operating System.

2. Discover application components hosted by each role. This is typically performed with WMI or script discoveries targeting the hosting role class.

3. Populate groups. This is typically performed with group population discoveries targeting the group classes themselves.

### Application Roles

The first step in the discovery process for an application model with multiple classes is typically to identify which agents the application is installed on. Because application installations are identified by the application role classes based on the Windows Computer Role class or the Windows Local Application class, these are typically the first classes discovered.

These discoveries will typically target a broad class, such as Windows Operating System, because the application could be installed on any computer in the environment. There is no way to know which computers the application is installed on until the discovery runs. If the application can only be installed on a particular version of Windows or if it can only be installed on a server or a client operating system, a slightly more refined target could be used. Examples include the Windows Server Operating System class for an application that could only be installed on a server or the Windows Server 2008 class for an application that could only be installed on a computer that is running Windows Server 2008.

Because discoveries of application roles are targeted at such a broad class, only registry discoveries should be used because they consume minimal resources. A heavier kind of discovery, such as a script, would put an unnecessary load on multiple computers that do not have an instance of the class being discovered. For most applications, a registry key is sufficient to at least confirm that the application exists. If certain properties must use another data source, such as a script, an additional discovery for the same class could be added that targeted to the class itself. This would guarantee that the more resource-intensive process runs only on those agents where it is required.

### Application Components

After the application’s roles are discovered, the newly discovered objects can be used as targets for discoveries identifying application components. These classes are typically hosted by one of the application’s roles so that the hosting parent class can be used for the target. In addition to making sure that the discovery only runs on those agents where instances of the class being discovered are likely to be located, using the hosting parent as a target provides access to the hosting parent class’s properties by using $Target variables.

For example, suppose that a class named MyLocalApplication hosts a class called MyApplicationComponent. When an instance of MyApplicationComponent is discovered, it must provide a value for any key properties of MyLocalApplication because that is the class’s hosting parent. If the discovery for MyApplicationComponent targets MyLocalApplication, the discovery has access to all the properties of MyLocalApplication by using $Target variables.

Discoveries for application components typically use either WMI or script discovery. Registry discovery can be used if information about the application is stored there, but frequently another data source is required for this information. If a WMI provider for the application is available and the discovery does not require complex logic, a WMI discovery can be used. If more complex logic is required, a script can be used to access information from the registry, WMI, or any other data source that can be accessed from a script.

### Group Population

After all other classes are discovered, group population may be performed because this discovery uses instances and properties that have already been discovered. Group population discoveries target the group class itself. Because groups are unhosted, they are managed by the Root Management Server. Any discoveries targeted at a group class will run on the RMS. This is appropriate because this is where group population discoveries are required to run so that the discovery can access information in the Operations Manager database.

## Discovery Intervals

Most discoveries are run at set intervals, and this interval must be specified when the discovery is created. If the interval is too long, the user may have to wait an extended time until new instances or changes to existing instances are discovered. For example, a particular discovery for an application may be configured to run one time every 24 hours. When a new copy of the application is installed on a computer, it may take that long until the new instance appears in Operations Manager 2007 and monitoring begins.

Too short a time interval puts too much overhead on agents while providing minimal value. Also, if instances of the discovered class are expected to change frequently enough to warrant an aggressive discovery schedule, the design of the class may violate the best practices for classes as described here.

The following table provides recommended intervals for different kinds of discoveries.

|  |  |  |
| --- | --- | --- |
| Discovery Type | Minimum Interval | Comments |
| Registry | 5 minutes | The interval for registry discoveries can be set very low with minimal effect, assuming that properties of the discovered objects rarely change. Because registry discoveries typically discover classes that change so rarely however, an interval of as much as 24 hours is often used. |
| WMI Query | 4 hours | Avoid the use of wildcard characters (\*) in the query to return all properties. Should not be targeted at a broad class, such as Microsoft Windows Operating System. |
| Script | 4 hours | Should not be targeted at a broad class, such as Microsoft Windows Operating System. |

## See Also

[Discovery](#zbe7560f04e134efd8884bcbb11d5bf3c)

[Creating Discoveries](#z6014eb924a6d440aa079a22814740bd5)

# Practices to Avoid

The following are common mistakes that are made when you design a service model for an application.

## Classes

### Creating Too Many Classes

Creating too many classes can result in needless complexity with minimal value. A good rule is to use the least number of classes to achieve the desired monitoring results. Other than abstract classes, if a class is not going to be the target of any rules or monitors, it probably should not be created. Also, if two application components are similar, consider modeling them with a single class, possibly by using a property that can hold the values for any differences.

### Creating Classes for Volatile Objects

Classes should represent objects that are fairly static. They are created and removed by administrators of the managed computers and not created frequently through automated processes. Examples include an application installed on a server, a database, and a Web site. Generally, a new instance of a class should not be expected most of the times that a discovery for that class runs. Too much volatility can result in excess load on the Root Management Server and lead to error messages from too frequent updates to the Operations Manager database. Other than performance considerations, it is also typically not useful to have health measurements for objects that appear for a short time.

For example, consider a Voice over IP application. You may want to create a class representing a telephone call. A monitor targeted at the class would measure the quality of each call and report an error state if there is a decrease in the quality level of the call. This is a questionable design, because telephone calls are constantly created and destroyed. In fact, discovery for the class would have to be run very frequently; otherwise, most of the calls would never be discovered. For those calls that were discovered, by the time that an administrator was able to respond to a message generated by a monitor, the telephone call would likely have ended.

A better solution for this scenario would be to target monitors at a class created for the application installation on the server. It might be used as a target for a monitor measuring the health of calls and detecting any error events that indicate a problem with a particular call. Any health state would be associated with the application itself, and any alerts could include an identifier for a particular call should a problem are detected.

### Properties that Update Too Frequently

Properties should change rarely after they are first discovered. After initial discovery of an object, discovery data is sent from the agent to the Root Management Server only when a configuration change is detected. If properties frequently change, discovery data is sent to the Root Management Server every time that discovery is run and results in excess load. Frequent updating can also affect configuration reports that detect changes in property values.

The most common cause of this issue is a model that stores performance data or health state in a property, which is not a recommended practice. For example, if an object performs some kind of replication, you might consider creating a property to hold the last replication time. But this kind of performance data should not be stored in a property. A better solution would be to collect a numeric value, such as the number of minutes or hours since the last replication, as performance data. This data could be collected with a rule for tracking it on a graph in the console or in a report. A monitor could also be created comparing the collected performance value with a numeric threshold to set the health state according to the time since the last successful replication.

## Discoveries

### Too Frequent Discoveries

New objects installed on a managed computer or changes to existing objects will not be recognized by Operations Manager until the next time that the discovery for the class runs. Frequencies of only a few minutes are sometimes used to reduce the time that is required for such additions and changes to be detected. However, a short frequency interval increases the load on the agent running the discovery and usually provides minimal value.

Even though a particular management pack may have only a few discoveries, an agent must run discoveries from multiple management packs installed in the environment in addition to the other kinds of workflows, such as rules and monitors. Because of this, care should be taken to balance the need for quick detection of changes with a minimization of overhead.

If a particular class requires a short discovery interval because of frequent changes to the application’s components, this may indicate a health model that violates the above recommendations for volatile objects and properties that update frequently.

### Script Discoveries Targeting Broad Classes

Discoveries for some classes in a management pack will need to search across all computers in an environment (or at least a large set of computers) to identify where the application is installed. These discoveries will continue to run on a regular basis even though the application may never be installed on most of these computers.

Because of their fairly small overhead, we strongly recommended that only registry discoveries target broad classes, such as the Windows Operating System class. Script discoveries that have significantly larger resource requirements should target only those classes specific to the application that were previously discovered through the registry. This strategy means that only those computers that have the application installed are required to run the scripts. Computers without the application installed must run only registry discoveries. These have minimal resource requirements.

## See Also

[Defining Classes and Relationships](#z039ae803443b49a5a8281b2197e915c8)

[Defining Discoveries](#ze8857ff93cf24bcf90913ae6ce2913b5)

# Building a Service Model

The following sections provide walkthroughs of the activities that support the implementation of a service model.

## In This Section

[Creating Classes and Relationships](#ze2ee647ce5bb41dca88a5cc693cf602c)

|  |
| --- |
| Create different types of classes and relationships by using the Operations Manager Authoring console. |

[Creating Discoveries](#z6014eb924a6d440aa079a22814740bd5)

|  |
| --- |
| Create different types of discoveries by using the Operations Manager Authoring console. |

# Creating Classes and Relationships

The following procedures provide guidance on how to create classes and relationships in the Operations Manager 2007 Authoring console.

## In This Section

[How to Create a New Management Pack](#z69828350433546e78e90ea9fa37b1703)

|  |
| --- |
| Create a new management pack by using the Authoring console. |

[How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596)

|  |
| --- |
| Create an abstract class, computer role classes, and application components. |

[How to Create a Group](#zacbb87d60b9e49c389b27b3d57083cee)

|  |
| --- |
| Create different kinds of group classes. |

[How to Create a Containment Relationship](#z8d524c60568f4c37a7ed8c9e86515c32)

|  |
| --- |
| Create containment relationships for health rollup and for group membership. |

# How to Create a New Management Pack

The following procedure describes how to create a new management pack by using the Authoring console.

To create a new management pack with the Operations Manager 2007 Authoring console

|  |
| --- |
| 1. Start the Authoring console.  2. Select File, and then select New.  3. On the Management Pack Template page, in the Select a Management Pack Template box, select Empty Management Pack.  4. In the Management Pack Identity box, type MyMP.  5. Click Next.  6. On the Name and Description page, in the Display Name box, type My Management Pack.  7. Click Create.  8. Select File, and then click Save.  9. Make sure that the File Name box has MyMP.xml, and then click Save. |

# How to Create a Class

The following procedures explain how to create classes in the Operations Manager 2007 Authoring console. Before you perform these procedures, you must first complete the prerequisite procedure, [How to Create a New Management Pack](#z69828350433546e78e90ea9fa37b1703), in which you first create a new management pack before adding the classes shown here.

The classes created in the following procedures have these characteristics:

 Abstract class based on Windows Computer Role named MyMP.MyComputerRoleBase. Class has a single non-key property named Version and no key property.

 Two classes based on the abstract class named MyComputerRole1 and MyComputerRole2.

 Single application component hosted by a computer role called MyApplicationComponent. Class has a single key property named ComponentName.

To create an abstract class

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Classes.  2. Right-click in the Classes pane, select New, and then select Windows Server role.  3. In the ID box, type the name MyMP.MyComputerRoleBase.  4. In the Display Name box, type My Computer Role Base.  5. Click Finish.  6. Right click MyMP.MyComputerRoleBase and select Properties.  7. On the General tab, in the Attributes pane, select Abstract.  8. On the Properties tab, do the following:  a. Right-click in the navigation pane and select Add Property.  b. In the Choose a unique identifier box, type Version, and then click OK.  c. In the Display Name box, type Version. Because you are expecting only a single instance of this class on a computer, it does not require a key property.  d. Click OK.  9. Select File, and then click Save. |

To create classes based on a custom abstract class

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Classes.  2. Right-click in the Classes pane, select New, and then select Custom Class.  3. In the Choose a unique identifier box, type the name MyMP.MyComputerRole1, and then click OK.  4. On the General tab, do the following:  a. In the Base Class box, select MyMP.MyComputerRoleBase.  b. In the Display Name box, type My Computer Role 1. Click OK.  5. Right-click in the Classes pane, select New, and then select Custom Class.  6. In the Choose a unique identifier box, type the name MyMP.MyComputerRole2, and then click OK.  7. On the General tab, do the following:  a. In the Base Class box, select MyMP.MyComputerRoleBase.  b. In the Display Name box, type My Computer Role 2. Click OK.  8. Select File, and then click Save. |

To create an application component

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Classes.  2. Right-click in the Classes pane, select New, and then select Windows Local Application Component.  3. On the General page, do the following:  a. In the ID box, type the name MyMP.MyApplicationComponent.  b. In the Display Name box, type My Application Component.  c. Click Next.  4. On the Key Properties page, do the following:  a. Check the Specify a key property box. You are expecting multiple instances of MyApplicationComponent for each instance of MyApplication, which is its hosting parent. Therefore, the class requires a key property.  b. In the ID box, replace the existing text with ComponentName.  c. In the Type box, select string.  d. In the Name box, type Component Name. Click Next.  5. On the Hosting Class page, do the following:  a. In the Hosting Class box, type MyMP.MyComputerRole1.  b. In the Hosting Relationship ID box, type MyMP.MyComputerRole1HostsMyApplicationComponent.  c. In the Hosting Relationship Name box, type My Computer Role 1 Hosts My Application Component. Click Finish.  6. Select File, and then click Save. |

## See Also

[Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae)

[Choosing a Base Class](#z6709d79030034a5680b92a3bd9815bd6)

# How to Create a Group

The following procedures explain how to create classes for an instance group, a computer group, and a health rollup in the Operations Manager 2007 Authoring console. Before you perform these procedures, you must first complete the prerequisite procedure, [How to Create a New Management Pack](#z69828350433546e78e90ea9fa37b1703), in which you first create a new management pack before adding the classes shown here.

To add a reference for Instance Group Library

|  |
| --- |
| 1. Select File, and then select Management Pack Properties.  2. Click the References tab.  3. Click Add Reference.  4. Move to the folder where the Authoring console is installed, and select Microsoft.SystemCenter.InstanceGroup.Library.mp.  5. Click in the Alias box, and change the alias to SCIG. Click OK. Any alias can be used for the alias. A short string is typically most convenient. |

To create an instance group

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Classes.  2. Right-click in the Classes pane, select New, and then select Custom Class.  3. In the Choose a unique identifier… box, type the name MyMP.MyInstanceGroup, and then click OK.  4. On the General tab, do the following:  a. In the Base Class box, select Browse all classes.  b. In the Management Pack Class Chooser box, select Microsoft.SystemCenter.InstanceGroup, and then click OK.  c. In the Name box, type My Instance Group.  5. In the Attributes pane, select Singleton. Click OK.  6. Select File, and then click Save. |

To create a computer group

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Classes.  2. Right-click in the Classes pane, select New, and then select Computer Group.  3. In the ID box, type the name MyMP.MyComputerGroup.  4. In the Display Name box, type My Computer Group.  5. Click Finish.  6. Select File, and then click Save. |

To create a group for health rollup

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Classes.  2. Right-click in the Classes pane, select New, and then select Custom Class.  3. In the Choose a unique identifier box, type the name MyMP.HealthRollup, and then click OK.  4. On the General tab, do the following:  a. In the Base Class box, select Browse all classes.  b. In the Management Pack Class Chooser box, select System.ComputerRole.  c. In the Name box, type My Health Rollup. Click OK.  5. In the Attributes pane, select Singleton. Click OK.  6. Select File, and then click Save. |

## See Also

[Classes](#zeeceb3374daf4cfcaf6fdfdebeff11ae)

[Defining Classes and Relationships](#z039ae803443b49a5a8281b2197e915c8)

# How to Create a Containment Relationship

The following procedures show how to create containment relationships in the Operations Manager 2007 Authoring console. Before you perform these procedures, you must complete the following prerequisite procedures:

 [How to Create a New Management Pack](#z69828350433546e78e90ea9fa37b1703) - Create the management pack in which the relationships will be created.

 [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) - Create classes that will serve as both the source and target in relationships.

 [How to Create a Group](#zacbb87d60b9e49c389b27b3d57083cee) - Create groups class that will serve as the source in a relationship.

The relationships created in the following procedures have these characteristics.

 Relationship between MyHealthRollup group class and MyComputerRole1 class. This supports a health rollup of all instances of MyComputerRole1.

 Relationship between MyComputerRole2 and MyComputerRole1. This supports a health dependency of MyComputerRole2 on MyComputerRole1.

Containment relationships do not have to be created for Instance Groups or Computer Groups, because the required relationships are already defined by the classes on which these classes are based.

To create a containment relationship for a health rollup

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Relationships.  2. Right-click in the Relationships pane, select New, and then select Containment Relationship.  3. On the General page, do the following:  a. In the ID box, type MyMP.MyHealthRollupContainsMyComputerRole1.  b. In the Display Name box, type My Health Rollup Contains My Computer Role 1.  c. Click Next.  4. On the Source and Target page, do the following:  a. In the Class ID (Source) box, select MyMP.MyHealthRollup.  b. In the Class ID (Target) box, select MyMP.MyComputerRole1.  c. Click Finish.  5. Select File, and then click Save. |

To create a containment relationship for dependency between two classes

|  |
| --- |
| 1. In the Authoring console, select Service Model, and then select Relationships.  2. Right-click in the Relationships pane, select New, and then select Containment Relationship.  3. On the General page, do the following:  a. In the ID box, type MyMP.MyComputerRole2ContainsMyComputerRole1.  b. In the Display Name box, type My Computer Role 2 Contains My Computer Role 1.  c. Click Next.  4. On the Source and Target page, do the following:  a. In the Class ID (Source) box, select MyMP.MyComputerRole2.  b. In the Class ID (Target) box, select MyMP.MyComputerRole1.  c. Click Finish.  5. Select File, and then click Save. |

## See Also

[Relationships](#z332a9f5d49484136be5a6f3fc0b85218)

# Creating Discoveries

The following procedures provide guidance on how to create classes and relationships in the Operations Manager 2007 Authoring console.

## In This Section

[How to Create a Registry Discovery](#zd4f3854d9d9c4fe5bccebe86c7b83f08)

|  |
| --- |
| Create a discovery based on keys and values in the registry. |

[How to Create a WMI Discovery](#zf3b86ce3197849df888d30f21fb192d1)

|  |
| --- |
| Create a discovery based on a WMI query. |

[How to Create a Script Discovery](#z9edec4621dbe41e58335828beeee6e98)

|  |
| --- |
| Create a discovery based on a script written in VBScript. |

[How to Create a Windows PowerShell Discovery](#z7fec9a7b629a409e81cd12d8b5c9d7a8)

|  |
| --- |
| Create a discovery based on a script written in PowerShell. |

[How to Populate a Group](#zfdbcbe90485b4f81bc21c12b5268ad38)

|  |
| --- |
| Create discoveries to populate different kinds of groups. |

# How to Create a Registry Discovery

The following procedure shows how to create a registry discovery in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the classes to be discovered.

The registry discovery created in this procedure has the following characteristics:

 Searches for the application on all computers.

 Discovers a class named MyComputerRole1. The class has a single non-key property named Version and has no key property.

 An instance of the class should only be created if a registry key named HKLM:\SOFTWARE\MyApp exists.

 If a class is created, the version should be collected from the value HKLM:\SOFTWARE\MyApp\Version.

To create a registry discovery

|  |
| --- |
| 1. Select Health Model, and then select Discoveries.  2. Right-click in the right-side pane, select New, and then select Registry Filtered.  3. On the General page, do the following:  a. In the Element ID box, type MyMP.Discovery.MyComputerRole1.  b. In the Display Name box, type Discover Computer Role 1.  c. In the Target box, select Browse all classes  d. In the Management Pack Class Chooser box, select Microsoft.Windows.OperatingSystem, and then click OK.  e. In the Category box, select Discovery, and then click Next.  f. Click Next.  4. On the Schedule page in the Run every box, select 1 Hours, and then click Next. Do not select the box Synchronize at: box  5. On the Computer page, notice that the Computer box should already contain the following required text: $Target/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/NetworkName$. If it does not, instead of typing the text, fill it in by selecting the button to the right side of the Computer box, selecting (Host=Windows Computer), and then selecting Network Name (Windows Computer). Click Next. This option specifies the name of the computer on which the registry will be searched. The $Target variable specified refers to the network name of the computer that is hosting the target operating system object.  6. On the Registry Probe Configuration page, do the following:  a. Click Add.  b. In the Edit Attribute Properties dialog box, select Key for the Object Type.  c. In the Name box, type AppExists.  d. In the Path box, type SOFTWARE\MyApp.  e. In the Attribute Type box, select Check if exists. Click OK.  f. Click Add.  g. In the Edit Attribute Properties dialog box, select Value for the Object Type.  h. In the Name box, type AppVersion. In the Path box, type SOFTWARE\MyApp\Version.  i. In the Attribute Type box, select String. Click OK.  j. Click Next.  7. On the Build Event Expression page, do the following:  a. Click Insert.  b. Click the … button to the right side of the Parameter Name box, and then select AppExists.  c. In the Operator box, select Equals.  d. In the Value box, type True.  e. Click Next.  8. On the Discovery Mapper page, do the following:  a. In the Class ID box, select MyMP.ComputerRole1.  b. In the Key Properties section, click the button to the right of the Value for Microsoft.Windows.Computer/PrincipalName, select (Host=Windows Computer), and then select Principal Name (Windows Computer). This step fills in the following text for the value: $Target/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/PrincipalName. This is a $Target variable that resolves to the computer name of the target agent.  c. In the Non-Key Properties section, in the Value for MyMP.MyComputerRoleBase\Version, type $Data/Values/AppVersion$. This step uses a $Data variable to resolve to the data that is stored in the Version registry value that is defined on the Registry Probe Configuration page. It cannot be selected from a menu because the menu can only prompt for $Target variables.  d. Click the button to the right side of the Value for System.Entity/DisplayName, select (Host=Windows Computer), and then select Principal Name (Windows Computer).  e. Click Finish.  9. Select File, and then click Save. |

# How to Create a WMI Discovery

The following procedure shows how to create a Windows Management Instrumentation (WMI) discovery in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the classes to be discovered.

The discovery created in this procedure has the following characteristics:

 Searches for the application on all computers that have an instance of MyComputerRole1.

 Discovers a class named MyComputerRole2. The class has a single non-key property named Version and no key property.

 An instance of the class should only be created if a file named C:\MyApp\MyApp.exe exists.

 If a class is created, the version should be collected from the version property on the file.

To create a WMI discovery

|  |
| --- |
| 1. Select Health Model, and then select Discoveries.  2. Right-click in the right-side pane, select New, and then select WMI.  3. On the General page, do the following:  a. In the Element ID box, type MyMP.Discovery.MyComputerRole2.  b. In the Display Name box, type Discover Computer Role 2.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Category box, select Discovery, and then click Next.  4. On the WMI Configuration page, do the following:  a. In the WMI Namespace box, type root\cimv2.  b. In the Query box, type select name, version from cim\_datafile where name ='c:\\MyApp\\MyApp.exe'.  c. In the Frequency (seconds) box, type 14400. Click Next.  5. On the Discovery Mapper page, do the following:  a. In the Class ID box, select MyMP.ComputerRole2.  b. In the Key Properties section, click the button to the right side of the Value for Microsoft.Windows.Computer/PrincipalName, select (Host=Windows Computer), and then select Principal Name (Windows Computer). This step fills in the following text for the value: $Target/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/PrincipalName. This is a $Target variable that resolves to the computer name of the target agent.  c. In the Non-Key Properties section, in the Value for MyMP.MyComputerRoleBase\Version, type $Data/Property[@Name='Version']$. This step uses a $Data variable to resolve to the data returned from the WMI query. It cannot be selected from a menu because the menu can only prompt for $Target variables.  d. Click the button to the right side of the Value for System.Entity/DisplayName, select (Host=Windows Computer), and then select Principal Name (Windows Computer).  e. Click Finish.  6. Select File, and then click Save. |

## See Also

[Discovery](#zbe7560f04e134efd8884bcbb11d5bf3c)

# How to Create a Script Discovery

The following procedure shows how to create a script discovery in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the classes to be discovered.

The discovery created in this procedure has the following characteristics:

 Searches for class on any agent with an instance of MyComputerRole1.

 Discovers three instances of a class named My Application Component. The class has a single key property called ComponentName and a non-key property named Version.

 The discovered class is hosted by a class named MyComputerRole1 which is hosted by the Windows Computer class. This hosting class has no key property.

To create a script discovery

|  |
| --- |
| 1. Select Health Model, and then select Discoveries.  2. Right-click in the Discoveries pane, select New, and then select Script.  3. On the General page, do the following:  a. In the ElementID box, type MyMP.Discovery.MyApplicationComponent.  b. In the Display Name box, type Discover MyApplicationComponent.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Category box, select Discovery. Click Next.  4. On the Schedule page, in the Run every: box, type 4 hours. Click Next.  5. On the Script page, do the following:  a. In the File Name box, type DiscoverApplicationComponents.vbs.  b. In the Timeout box, type 5 minutes.  c. Copy the complete following script, and paste it into the Script box.  SourceId = WScript.Arguments(0)  ManagedEntityId = WScript.Arguments(1)  sComputerName = WScript.Arguments(2)  Set oAPI = CreateObject("MOM.ScriptAPI")  Set oDiscoveryData = oAPI.CreateDiscoveryData(0, SourceId, ManagedEntityId)  For i = 1 to 3  Set oInstance = oDiscoveryData.CreateClassInstance("$MPElement[Name='MyMP.MyApplicationComponent']$")  oInstance.AddProperty "$MPElement[Name='Windows!Microsoft.Windows.Computer']/PrincipalName$", sComputerName  oInstance.AddProperty "$MPElement[Name='MyMP.MyApplicationComponent']/ComponentName$", "Component" & i  oDiscoveryData.AddInstance(oInstance)  Next  oAPI.Return(oDiscoveryData)  d. Click Parameters.  e. In the Parameters box, type $MPElement$, followed by a space.  f. Click Target, and select Id.  g. Type a space after $Target/Id$.  h. Click Target, select (Host=Windows Computer), and then select Principal Name (Windows Computer). Make sure that there is a space between the three variables in the Parameters box.  i. Click OK, and then click Finish.  6. Select File, and then click Save. |

# How to Create a Windows PowerShell Discovery

The following procedure shows how to create a Windows PowerShell discovery in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the classes to be discovered. This procedure implements the same discovery as in the procedure [How to Create a Script Discovery](#z9edec4621dbe41e58335828beeee6e98) by using Windows PowerShell instead of VBScript. The Operations Manager Authoring console does not include a wizard for creating this kind of discovery. Therefore, a custom discovery must be created.

The discovery has the following characteristics:

 Searches for a class on any agent with an instance of MyComputerRole1.

 Discovers three instances of a class named MyApplicationComponent. The class has a single key property called ComponentName and a non-key property named Version.

 The discovered class is hosted by a class named MyComputerRole1 which is hosted by Windows Computer. This hosting class has no key property.

To create a Windows PowerShell discovery

|  |
| --- |
| 1. Select Health Model, and then select Discoveries.  2. Right-click in the Discoveries pane, select New, and then select Custom Discovery.  3. In the Choose a unique identifier box, type MyMP.Discovery.MyApplicationComponent.PowerShell.  4. On the General page, do the following:  a. In the Name box, type Discover Application Components with PowerShell.  b. In the Target box, select MyMP.MyComputerRole1.  5. On the Discovered Classes page, do the following:  a. Click Add above the Discovered classes and their attributes pane, and select Add discovered type.  b. In the Management Pack Class Chooser box, select MyMP.MyApplicationComponent, and then click OK.  6. On the Configuration page, do the following:  a. Select Browse for a type.  b. In the Choose module type box, select Microsoft.Windows.TimedPowerShell.DiscoveryProvider.  c. In the Module ID box, type PSScript. Click OK.  d. In the IntervalSeconds box, type 14400. This is the equivalent of 4 hours.  e. In the SyncTime box, clear the existing text.  f. In the ScriptName box, type DiscoverMyApplicationComponent.ps1.  g. In the TimeoutSeconds box, type 300. This is the equivalent of 5 minutes.  h. Click Edit. This starts the external editor.  i. Paste the complete contents of the following script between the ScriptBody tags in the XML. Replace any text that might already exist.  <![CDATA[  param($sourceId,$managedEntityId,$computerName)  $api = new-object -comObject 'MOM.ScriptAPI'  $discoveryData = $api.CreateDiscoveryData(0, $SourceId, $ManagedEntityId)  for ($i=1; $i -le 3; $i++)  {  $instance = $discoveryData.CreateClassInstance("$MPElement[Name='MyMP.MyApplicationComponent']$")  $instance.AddProperty("$MPElement[Name='Windows!Microsoft.Windows.Computer']/PrincipalName$", $computerName)  $instance.AddProperty("$MPElement[Name='MyMP.MyApplicationComponent']/ComponentName$", 'Component' + $i)  $discoveryData.AddInstance($instance)  }  $discoveryData  ]]>  j. Add the following XML after the ScriptBody tags and before the TimeoutSeconds tags:  <Parameters>  <Parameter>  <Name>sourceID</Name>  <Value>$MPElement$</Value>  </Parameter>  <Parameter>  <Name>managedEntityID</Name>  <Value>$Target/Id$</Value>  </Parameter>  <Parameter>  <Name>computerName</Name>  <Value>$Target/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/PrincipalName$</Value>  </Parameter>  </Parameters>  k. The complete configuration of the discovery in the external editor should now resemble the following:  <Configuration p1:noNamespaceSchemaLocation="C:\Users\Username\AppData\Local\Temp\MyMP.Discovery.MyApplicationComponent.PowerShell.xsd" xmlns:p1="http://www.w3.org/2001/XMLSchema-instance">  <IntervalSeconds>14400</IntervalSeconds>  <SyncTime></SyncTime>  <ScriptName>DiscoverMyApplicationComponent.ps1</ScriptName>  <ScriptBody>  <![CDATA[  param($sourceId,$managedEntityId,$computerName)  $api = new-object -comObject 'MOM.ScriptAPI'  $discoveryData = $api.CreateDiscoveryData(0, $SourceId, $ManagedEntityId)  for ($i=1; $i -le 3; $i++)  {  $instance = $discoveryData.CreateClassInstance("$MPElement[Name='MyMP.MyApplicationComponent']$")  $instance.AddProperty("$MPElement[Name='Windows!Microsoft.Windows.Computer']/PrincipalName$", $computerName)  $instance.AddProperty("$MPElement[Name='MyMP.MyApplicationComponent']/ComponentName$", 'Component' + $i)  $discoveryData.AddInstance($instance)  }  $discoveryData  ]]>  </ScriptBody>  <Parameters>  <Parameter>  <Name>sourceID</Name>  <Value>$MPElement$</Value>  </Parameter>  <Parameter>  <Name>managedEntityID</Name>  <Value>$Target/Id$</Value>  </Parameter>  <Parameter>  <Name>computerName</Name>  <Value>$Target/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/PrincipalName$</Value>  </Parameter>  </Parameters>  <TimeoutSeconds>300</TimeoutSeconds>  </Configuration>  l. Close the external editor, and save the configuration in the Authoring console.  7. Click OK.  8. Select File, and then click Save. |

## See Also

[Discovery Scripts](#z874110d432734565a5ee6eaff0826a34)

# How to Populate a Group

The following procedures show how to create a discovery to populate groups in the Operations Manager 2007 Authoring console. Before you perform these procedures, you must complete the following prerequisite procedures:

 [How to Create a New Management Pack](#z69828350433546e78e90ea9fa37b1703) - Create the management pack to contain the classes and groups.

 [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) - Create classes that will be contained in the groups.

 [How to Create a Group](#zacbb87d60b9e49c389b27b3d57083cee) - Create groups that will be populated.

 [How to Create a Containment Relationship](#z8d524c60568f4c37a7ed8c9e86515c32) – Create relationships that will be discovered.

To populate an instance group

|  |
| --- |
| 1. Select Health Model, and then select Discoveries.  2. Right-click in the right side pane, select New, and then select Custom Discovery.  3. In the Choose a unique identifier box, type MyMP.Discovery.Populate MyInstanceGroup.  4. On the General tab, do the following:  a. In the Name box, type Populate My Instance Group.  b. In the Target box, select MyMP.MyInstanceGroup.  5. On the Discovered Classes tab, do the following:  a. Click Add above the Discovered relationships and their attributes pane and select Add discovered relationship.  b. In the Management Pack Relationship Chooser box, select Microsoft.SystemCenter.InstanceGroupContainsEntities, and then click OK.  6. On the Configuration tab, do the following:  a. Select Browse for a type.  b. In the Choose module type box, select Microsoft.SystemCenter.GroupPopulator.  c. In the Module ID box, type GroupPopulator. Click OK.  d. In the Value column of the RuleId row, type $MPElement$.  e. In the Value column of the GroupInstanceId row, type $MPElement[Name="MyMP.MyInstanceGroup"]$.  f. In the Value column of the >MonitoringClass row, type $MPElement[Name="MyMP.MyComputerRole1"]$.  g. In the Value column of the >>RelationshipClass row, type $MPElement[Name="SCIG!Microsoft.SystemCenter.InstanceGroupContainsEntities"]$. If an alias other than SCIG is used, this string should be replaced with the appropriate alias.  7. Click OK.  8. Select File, and then click Save. |

To populate a computer group

|  |
| --- |
| 1. Select Health Model, and then select Discoveries.  2. Right-click in the right-side pane, select New, and then select Custom Discovery.  3. In the Choose a unique identifier box, type MyMP.Discovery.PopulateMyComputerGroup.  4. On the General tab, do the following:  a. In the Name box, type Populate My Computer Group.  b. In the Target box, select MyMP.MyComputerGroup.  5. On the Discovered Classes tab, do the following:  a. Click Add above the Discovered relationships and their attributes pane, and select Add discovered relationship.  b. Select Microsoft.SystemCenter.ComputerGroupContainsComputer. Click OK.  6. On the Configuration page, do the following:  a. Select Browse for a type.  b. In the Choose module type box, select Microsoft.SystemCenter.GroupPopulator.  c. In the Module ID box, type GroupPopulator. Click OK.  d. Click Edit to start the external editor.  e. Copy and paste the following XML code between the Configuration tags.  <RuleId>$MPElement$</RuleId>  <GroupInstanceId>$MPElement[Name="MyMP.MyComputerGroup"]$</GroupInstanceId>  <MembershipRules>  <MembershipRule>  <MonitoringClass>$MPElement[Name="Windows!Microsoft.Windows.Computer"]$</MonitoringClass>  <RelationshipClass>$MPElement[Name="SC!Microsoft.SystemCenter.ComputerGroupContainsComputer"]$</RelationshipClass>  <Expression>  <Contains>  <MonitoringClass>$MPElement[Name="MyMP.MyComputerRole1"]$</MonitoringClass>  </Contains>  </Expression>  </MembershipRule>  </MembershipRules>  f. Close the external editor, and save the configuration in the Authoring console.  g. Click OK.  7. Select File, and then click Save. |

# Health Model

In Operations Manager 2007, a health model defines the logic used to measure the operational health of the application represented by the classes defined in the service model. This includes monitors for measuring the health state of monitored objects, rules for collecting information for analysis and reporting, and tasks for users to perform on demand activities in their daily operations activities.

The health model is designed after the service model is completed. The service model defines the classes that represent the application, whereas the health model defines how to measure the health of those classes.

The topics in this section provide an overview of the different elements that comprise the health model, a basic process for designing a health model for an application represented by an existing service model, and examples of using the Operations Manager 2007 Authoring console to build different health model elements.

## In This Section

[Key Concepts](#z968d5069d6e24272b059fa0a66ea2067)

|  |
| --- |
| Explains the concepts of the different components in a health model. |

[Health Model Design Overview](#z1360d77d18064e85aac24aee6a1bc8f3)

|  |
| --- |
| Explains the process and best practices when you design a health model for a particular application. |

[Building a Health Model](#z1de16bb513ac4e90be911543c19f4f70)

|  |
| --- |
| Demonstrates the different tasks used for building a health model for a sample application that uses the Authoring console. |

# Key Concepts

A Health Model consists of different kinds of management pack elements that each perform a specific function. The following sections are not yet complete. When they are completed, they will describe the different kinds of management pack elements that are included in a health model.

## In This Section

Monitors and Health State

|  |
| --- |
| Health state represents the current health of the application, and the health of each class is tracked for historical analysis. Monitors measure the health state of a class in the application’s service model. This section defines the different kinds of monitors and how multiple monitors are used together to measure the overall application health. |

Rules

|  |
| --- |
| Rules collect monitoring data for online analysis and reporting, generate alerts that are not associated with health state, or run a command on a schedule. This section provides the details of each kind of rule and when they should be created. |

Tasks

|  |
| --- |
| Tasks are commands that can be run on demand by the user to collect information or perform some action on one or more agents. This section defines the different kinds of tasks and shows how they are defined. |

Monitoring Scripts

|  |
| --- |
| Scripts are used by different monitoring elements to perform complex functionality. This section provides details how monitoring scripts are written and how their output is used in a workflow. |

# Targets

Every monitor, rule, task, and discovery in System Center Operations Manager 2007 must have a target class. This is typically a class that is defined in the service model for the application but may be another general class such as Windows Operating System.

## Effects of a Target

The target of a workflow has the following effects:

 Which agents the management pack is delivered to

 Which agents the workflow will run on and how many copies of it will be loaded

 Which target variables are available to the workflow

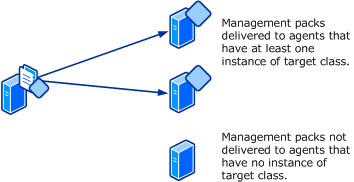
 Which object health state, alerts, and collected data will be associated with

Each of these topics is discussed in detail in the following sections:

### Which agents the management pack is delivered to

When a management pack is installed or changed, it is delivered to any agent that manages at least one instance of a class that is used as a target on at least one included workflow. Workflows targeted at classes that have instances on the agent are not loaded. However, the complete management pack must be delivered to the agent.

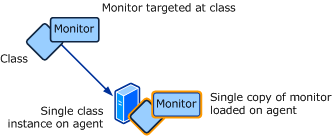
Management pack delivery



### Which agents the workflow will run on and how many copies of it will be loaded

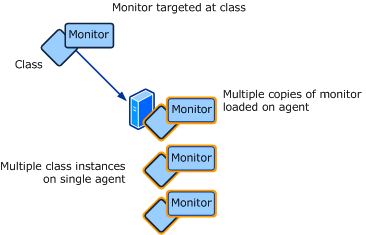
As soon as the management pack is deployed, the agent will load a separate copy of each workflow for each instance of its target class. This means that if there are multiple instances of the target class on the agent, then the agent will load multiple copies of the same workflow. Workflows included in the management pack that are targeted at objects not managed by the agent will not be run by that particular agent.

Workflow targeted at a single instance



The number of copies of the workflow that can be expected on any given agent must be accounted for when you define the workflow. Each copy of the workflow runs independently and includes its own set of variables. If a particular rule or monitor can potentially have multiple copies running on a single agent, then it has the potential to generate multiple alerts or collect multiple pieces of data when only a single one is desired.

Workflow targeted at multiple instances



For example, a rule may be defined to generate an alert in response to a particular event in the Windows event log. A common strategy for this scenario is to target a single instance class based on Windows Computer Role or Windows Local Application and use the Event Source and Event Number to specify the desired event. This typically works as expected because there will always be only a single instance of the target class on any agent. In this case, only a single copy of the rule is loaded, and the Event Source and Event Number are sufficient criteria to uniquely identify the event. If the same rule targets a class that has multiple instances on an agent, however, a separate alert would be created for each copy of the rule in response to the single event. In this second case, a $Target variable would have to be included in the rule’s expression to specify which copy of the workflow should generate the alert.

To show this example, the SQL Server 2008 Monitoring management pack includes a rule named Script : Failed to login that indicates that a monitoring script could not log on to a particular database. It determines this error has occurred by a particular event in the Operations Manager event log. This rule is targeted at the Microsoft SQL Server 2008 Database class which can have multiple instances on a particular agent – one for each database on that server. If only the Event Number and Event Source were specified, then a separate alert would be created for each database because the criteria in each copy of the rule would match the details specified for the event. In order to specify which copy of the rule should generate the alert, the expression in the rule also specifies that the Event Description must include the name of the database. It does this by using the DatabaseName property of the target object. Each copy of the rule resolves this variable when they are loaded by the agent so that only the copy of the rule matching the database name that is specified in the event will generate an alert.

### Which target variables are available to the workflow

A workflow has access to $Target variables for the properties of its target class and any of that class’s parent classes. If the workflow is targeted at a class based on Windows Computer Role for example, then properties of the Windows Computer class such as PrincipalName and IPAddress are available to the workflow since Windows Computer hosts Windows Computer Role.

### Which object health state, alerts, and collected data will be associated with

Any health state, alert, or performance data generated by a rule or monitor is associated with the object that the particular copy of the workflow was running against. In the earlier example, SQL 2008 DB Engine could have been used as the target because the rule would have been loaded on any agent running SQL Server 2008. The alert created from the rule though would be associated with the SQL Server itself has opposed to the individual database. This would have a significant affect any notifications, reports, or views specific to the particular object.

With a monitor, the target will determine which object’s health state will be affected. If a monitor measuring the health of a database were targeted at SQL 2008 DB Engine, the state of the SQL Server would be affected. However, the state of the database itself would not. An availability report would show that the server was unavailable for the period that the monitor was in a critical state, whereas the same report executed for the database itself would show no downtime. In this case, the monitor would have to be targeted at the SQL Server 2008 Database class in order to set the state for the intended object.

## Targeting Groups

Although groups can be the target of a workflow, this will most likely not result in the desired functionality. The workflow will not enumerate the contents of the group but will try to run against the group object itself. Because groups are managed by the Root Management Server, any workflows targeted at them will be loaded only on that server.

If you want to have a workflow run only against the members of a particular group, you can use the following method:

1. Create the rule or monitor with a target appropriate for the objects that are contained in the group.

2. Disable the rule or monitor on the Options tab of its Properties dialog box.

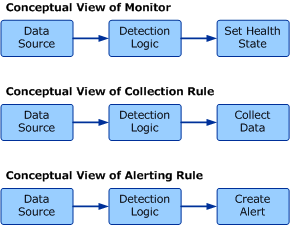
3. As soon as the rule or monitor is created, create an override to enable it for the group.

# Data Sources

Monitors and rules are the primary elements of a health model in System Center Operations Manager 2007 and provide similar yet distinct functionality. Monitors set the state of a monitored object while rules create simple alerts and collect data for analysis and reporting. Each monitor and rule is primarily defined by the source of the data that is used to perform its required functionality and the logic used to evaluate this data.

Monitors and rules are the primary elements of a health model and provide similar functionality. Monitors set the state if a monitored object, whereas rules create simple alerts and collect data for analysis and reporting. Each monitor and rule is defined by the source of the data that was used to perform its required functionality and the logic to evaluate this data.

Conceptual view of monitors and rules



Although they provide different functionality, monitors and rules use a common set of sources that provide the data to evaluate. For example, a monitor may use a performance counter to set the state of a particular object. A rule may access the same performance counter in order to store its value for analysis and reporting. The following sections provide the details of each different kind of data sources that may be used in monitors and rules.

Note

The data sources and monitoring logic discussed in this section are those that can be created by using wizards in the Authoring console. Other data sources and logic can be implemented by creating custom modules and workflows as discussed in the [Composition](#z94c023257fa84110b5234f4b8b2f927c) section of this guide.

## See Also

[Monitors and Health State](#z02191a4650bb4105bdc53793cc91984c)

[Rules](#zd2cda0b6b4cf4223b3048fb8df273167)

# Data Variables

Values of the data properties collected by a data source are accessed by $Data variables. The available properties and the syntax of $Data variables will be different for each kind of data source, and the syntax will be different for the same property in different usage. This is because all data that is used by a workflow is formatted in XML, and the syntax of the $Data variable is an XPath to a particular value. As different parts of the workflow process the data, it may be reformatted requiring a different XPath to the same value.

The following table lists common ways that $Data variables are used and the general syntax that is used to access each. The following sections provide the details of each kind of data source and the properties available for each. These sections provide additional details on the syntax that is specific to each kind of data.

|  |  |  |
| --- | --- | --- |
| Use | Description | Syntax |
| Criteria expressions | Criteria expressions do not require use of the $Data prefix to the variable or the dollar sign ($) delimiters. The syntax used for the parameter name in an expression will be the XPath to the property. For some data sources this is the name of the property. | Expressions are used to evaluate data to determine whether the data meets certain criteria. This may be inspecting an event to determine whether the data matches a particular source and number. Or, the expression might be comparing a performance value to a threshold. The $Data variable is used to identify the particular property from the data source being evaluated. |
| Mapping between data types | When you use them for mapping data, the syntax of the $Data variable will be the same as for a criteria expression except with the $Data prefix and the dollar sign ($) delimiters. This will typically have the following syntax:  $Data/<XPath to Property>$ | A data source may create data in one type that must be mapped to another type for the next step in a workflow. For example, collection rule may run a script that collects data to be stored as performance data. The data coming from the script would be formatted in a property bag and require mapping to performance data. The mapping would require the $Data variables to refer the values from the script’s property bag. |
| Alert Descriptions | The syntax for $Data variables in alert descriptions is different for alerts created by rules and alerts created by monitors. This is because an alert from a rule is in direct response to an expression while an alert from a monitor is response to a state change. Each may start with the same data source. However, the data is reformatted by the monitor after the state change.  $Data variables in alert descriptions from rules will typically have the following syntax:  $Data/<XPath to Property>$  $Data variables in alert descriptions from monitors will typically have the following syntax:  $Data/Context/<XPath to Property>$ | $Data variables can be used in alert descriptions to provide details of the data that created the alert. |

# Events

The following table lists the kinds of events that can be used for monitors and rules in a System Center Operations Manager 2007 management pack:

|  |  |
| --- | --- |
| Text Log | Text log file that has a single line per entry. |
| Text Log (Delimited) | Text log file that has a single line per entry and fields separated by a specific character. |
| Windows Events | Events in the Windows event log matching specified criteria. |
| WMI Events | Events created by Windows Management Instrumentation (WMI). |
| Syslog Events | Events from Unix systems and other devices. |

Details on each event data source are provided in the following sections. This includes the information that is required to retrieve the required data, properties available in the resulting data, and what workflows the event data source supports.

## Windows Events

Many Windows-based applications post information to events in a Windows event log. These events follow a standard format and frequently contain detailed information about the particular issue

### Criteria

In addition to the Application log to retrieve events from, workflows using a Windows event must specify sufficient criteria to identify the particular events that relate to the issue being identified. Frequently, the Event ID and the Event Source will be sufficient for this purpose. This depends on the kind of information that the application provides in the particular event in addition to the target that is being used for the monitor. If the class being used as the target for the monitor is expected to have multiple instances on a particular agent, then these two properties are probably insufficient for uniqueness. Unless the criteria included a key property for the target class then the criteria would possibly apply to all instances.

For example, the SQL Server 2008 (Monitoring) management pack uses an event to detect an error in a zone transfer. This event will have a publisher name of Microsoft-Windows-DNS-Server-Service and an event number of 6002. This monitor is called Microsoft.Windows.DNSServer.2008.Monitor.ZoneTransfer.ReinitializeZoneTransfer and uses the Microsoft.Windows.DNSServer.2008.Zone class for a target. This class is expected to have multiple instances on a particular agent. If multiple zones were hosted on a single DNS server, and only the publisher name and event number were used then the single event would be recognized by each instance. Each instance would set a negative health state even though the event was intended for only one zone. In this case, the name of the zone experiencing the issue is provided in parameter 2 of the event. Criteria is added to the monitor to match the name of the zone with the value in this event parameter to ensure that the criteria for each event matches only one instance of the Microsoft.Windows.DNSServer.2008.Zone class.

### Properties

The following table lists the properties available from Windows Events. These properties can be accessed for setting criteria in monitors and rules and can be included in alert descriptions.

|  |  |
| --- | --- |
| Property Name | Description |
| PublisherName | Source of the event. Generally used in the criteria of the monitor or rule. |
| Channel | Name of the event log such as Application or System. |
| LoggingComputer | Name of the computer logging the event. |
| EventNumber | Number of the event. |
| EventCategory | Category of the event. |
| EventLevel | Severity of the event that uses one of the following values.  0 - Success  1 - Error  2 - Warning  4 - Information |
| UserName | Name of the user account that was used to create the event. |
| EventDescription | Full event description. |
| Params | Collection of event parameters. |

### Workflows Supported

Windows Events can be used in the following kinds of workflows:

 Monitors

 Alerting Rules

 Collection Rules

## Text Logs

A text log is a text file that an application uses to log event information. In order to use a text log data source in a management pack, each entry in the log must be on a single line. If the log file does not fit this requirement, then a custom script has to be created to read the log. Using a standard text log, the monitor reads the whole line as a single entry. With a delimited text log, a delimiter is used to separate the fields in the entry.

Applications that use log files frequently create a new file each day or when one file reaches a certain size. To support this functionality, monitors and rules specify a Directory and a Pattern for the text logs being monitored. Directory is the path of the directory where the text logs will be located. This must be an absolute path without wildcard characters. A $Target variable could also be used if the path to the log files is stored in a property of the target class. Pattern is the name of the log file including wildcard characters as appropriate.

For example, an application might create a log file each day with the date included in the name as in log20100316.txt. A pattern for such a log might be log\*.txt which would apply to any log file following the application’s naming scheme.

### Properties

The expression for a text log monitor will include criteria that matches text in the log entry. For a standard text log this includes a search of the whole log entry treated as a single line. For a delimited text file, this will include a search of one or more of the included fields. The contents of a text log are included in the parameters of the event. For a standard text file, this is referenced by the parameter $Data/Params/Param[1]$. A delimited text file uses the same $Data variable by using the index number of the required parameter. The first field would be referenced with $Data/Params/Param[1]$, the second field would be referenced with $Data/Params/Param[2]$, and so on.

The following table lists the common properties available from text log monitors and rules:

|  |  |
| --- | --- |
| Property Name | Description |
| LogFileDirectory | Directory that the log file is located in. |
| LogFileType | CSV Log File Format for a delimited text log.  Generic Log File Format for text log. |
| LogFileName | Name of the log file that the event was taken from. |
| Param[1] | Complete entry in a standard text file. |
| Param[#] | Specific parameter in a delimited text file. # represents the number of the field. |

### Workflows Supported

Text Logs can be used in the following kinds of workflows:

 Monitors

 Alerting Rules

 Collection Rules

## WMI Events

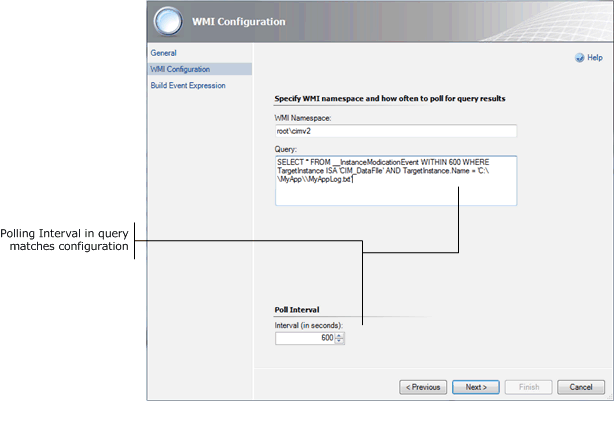
WMI events are created from WMI queries that detect particular actions in the operating system or in applications that create their own WMI events. These events can be used to detect such actions as a process ending, a file being created, or a registry key being modified. WMI events are not persisted. Therefore, any WMI events that are created when the agent service is not running are lost.

Note

This guide assumes knowledge of how to build a WMI notification query. For a an overview of this topic and sample queries see [Unlocking the Mystery of WMI Events in MOM](http://go.microsoft.com/fwlink/?LinkID=187607&clcid=0x409).

WMI monitors and rules require a namespace, a query, and a poll interval. The poll interval specified in the monitor or rule should match the poll interval in the query.

WMI matching poll intervals



### Criteria

Because criteria can be specified in the WHERE clause of the WMI query, criteria is frequently not required in the monitor or rule. It is only required if the query is expected to return multiple records.

Note

The Authoring Console wizards require that criteria be specified in WMI Event monitors and rules. If no criteria is required, then dummy criteria should be specified in the wizard and then removed by viewing the properties of the monitor or rule.

### Properties

The properties available for a WMI event will vary, depending on the kind of event being monitored. The properties available will also vary, depending on the properties of the WMI class included in the query. The data will be in the form of a property bag that has a collection of properties for one or more WMI class instances. WMI events created by using a query that uses either \_\_InstanceCreationEvent or \_\_InstanceDeletionEvent will have a single collection called TargetInstance with the instance being either created or deleted. WMI events created by using \_\_InstanceModificationEvent will have an additional collection called PreviousInstance.

The syntax for properties from a WMI event is as follows:

Collection[@Name='TargetInstance']/Property[@Name='Caption']

For example, the following WMI query monitors for the change in a file that is named c:\MyApp\MyAppLog.txt.

SELECT \* FROM \_\_InstanceNotificationEvent WITHIN 60 WHERE TargetInstance ISA 'CIM\_DataFIle' AND TargetInstance.Name = 'C:\\MyApp\\MyAppLog.txt'

Assuming that data is added to the file changing the file size and triggering the query, examples of properties from this query are shown in the following table:

|  |  |
| --- | --- |
| Property | Syntax |
| Original file size | $Data/Collection[@Name=’PreviousInstance’]/Property[@Name=’FileSize’]$ |
| New file size | $Data/Collection[@Name=’TargetInstance’]/Property[@Name=’FileSize’]$ |

### Workflows Supported

WMI events can be used in the following kinds of workflows:

 Monitors

 Alerting Rules

 Collection Rules

## Syslog Events

Syslog events can be used to collect messages from Unix systems and other devices. Syslog rules can be run on an agent that is the receiver of messages from one or more devices. When the rule is run, the agent will listen for messages on UDP port 514. This is the only port that can be used.

### Properties

The Syslog data properties are shown in the following table:

|  |  |
| --- | --- |
| Property Name | Description |
| Facility | The facility of the event that uses one of the values from the table that follows. |
| Severity | Numeric value that indicates the severity of the event using one of the following values:  0 - Emergency  1 - Alert  2 - Critical  3 - Error  4 - Warning  5 - Notice  6 - Info  7 - Debug |
| Priority |  |
| PriorityName |  |
| TimeStamp | Time that the message was sent. |
| HostName | Name of the device sending the message |
| Message | Text of the message |

#### Facility Values

The value for the facility property defines the part of the system that the message originated from. It will have one of the values from the following table:

|  |  |  |
| --- | --- | --- |
| Facility | Description | Value |
| 0 | Kernel | Kernel messages |
| 1 | User | User-level messages |
| 2 | Mail | Mail System |
| 3 | Daemons | System daemons |
| 4 | Auth | Security and authorization |
| 5 | Syslog | Syslog internal messages |
| 6 | LPR | Line printer subsystem |
| 7 | News | Network news |
| 8 | UUCP | Unix-to-Unix copy program |
| 9 | Cron | Cron daemon |
| 10 | Auth2 | Security and authorization |
| 11 | FTP | FTP daemon |
| 12 | NTP | Network time subsystem |
| 13 | LogAudit |  |
| 14 | LogAlert |  |
| 15 | Cron2 | Cron daemon |
| 16 | Local0 | Local use 0 |
| 17 | Local1 | Local use 1 |
| 18 | Local2 | Local use 2 |
| 19 | Local3 | Local use 3 |
| 20 | Local4 | Local use 4 |
| 21 | Local5 | Local use 5 |
| 22 | Local6 | Local use 6 |
| 23 | Local7 | Local use 7 |

### Workflows Supported

Syslog events can be used in the following kinds of workflows:

 Alert Rules

 Collection Rules

# Performance Data

Performance counters are numeric data that is used to measure the performance of some aspect of the application. The following table lists the kinds of data sources in a System Center Operations Manager 2007 management pack that produce performance data.

|  |  |
| --- | --- |
| Windows Performance | Samples a Windows performance counter at specified interval. |
| WMI Performance | Runs a WMI query at specified interval and uses the value of a numeric property for performance data. |

Details on each performance data source are provided in the following sections. This includes the information that is required to retrieve the required data, properties available in the resulting data, and what workflows the performance data source supports.

## Windows Performance

To define a monitor or rule based on a Windows performance counter, the object name and counter name of the performance counter to sample must be specified with a frequency that specifies how frequently to sample the data. The instance name only has to be specified if the same counter will be collected for multiple objects on the same agent. If this is the case, a $Target variable will typically be used for the value in the instance name in order to differentiate between the performance values for different objects. The counter must be available on the agent computer that is running the monitor or rule or an error will be created in the Operations Manager event log on the agent.

### Properties

The following table shows the properties that are available from a Windows performance counter.

|  |  |
| --- | --- |
| Property Name | Description |
| Object Name | Name of the performance object. |
| Counter Name | Name of the performance counter. |
| Instance Name | Name of the instance if it is specified. |
| Value | Numeric value of the performance data. |
| TimeSampled | Time that sample was performed in UTC format. |

### Workflows Supported

Windows performance can be used in the following kinds of workflows:

 Monitors

 Collection Rules

## WMI Performance

WMI performance refers to numeric data that is retrieved from a WMI query. This lets performance data be retrieved that is not available from a performance counter and without using the complexity and overhead of a script. The monitor or rule runs the query on a specified schedule and maps the value of the specified numeric field into the value property of the performance data.

For example, a monitor might have to track the size of a particular file. This might be a log file that indicates a particular problem when it exceeds a particular size. The name and size of the file could be retrieved from a query similar to the following:

Select Name, FileSize from CIM\_DataFile Where Name = 'C:\\MyApp\\MyAppLog.txt'

The monitor could run this query regularly by using the FileSize property for the value of the performance data and the Name property for the Instance property.

### Properties

The WMI query returns a property bag with properties returned from the query. This set of properties will vary, depending on the class returned and the properties specified in the query.

### Workflows Supported

WMI performance can be used in the following kinds of workflows:

 Monitors

 Collection Rules

## See Also

[Performance Monitors](#z7335542b6b38488aa703e6eaae902a46)

[Performance Collection Rules](#zceadc507cc0e4de39dac1bb6f3da3b52)

# Monitoring Scripts

Monitoring scripts are used when the required data cannot be collected through other standard means such as an event or performance counter. The script collects data from information on the agent and creates a property bag by using the MOM.ScriptAPI object that is installed with the Operations Manager 2007 agent.

Monitoring scripts may be written in any script language that can access the MOM.ScriptAPI object that is installed on all Operations Manager 2007 agents. The most common languages used are VBScript, JScript, and Windows PowerShell. Scripts that use VBScript or Jscript can be created by using modules in management pack libraries and wizards in the Authoring console. Modules are available for monitoring scripts that are written in Windows PowerShell in Operations Manager 2007 R2 but are not supported by wizards in the Authoring console. A custom workflow would have to be created by using the appropriate module as described in the Composition section of this guide.

## Property Bags

Monitoring scripts send any output data as a property bag so that it can be used by the rest of workflow. A property bag is a set of values that each has a name. Any name can be assigned although it is a best practice to use a name descriptive of the particular value. A property bag only exists during the life of the workflow. The next time that the workflow runs, the script is run and creates a new property bag with new values.

One property bag can have any number values, although the whole set of data may not exceed 4 MB. Most scripts will only require some values with a total size far under this limit. There is no requirement for all the values to be used by the workflow.

Scripts create property bags by using the CreatePropertyBag method on the MOM.ScriptAPI object. The workflow uses values from a property bag with a $Data variable that uses the following syntax:

$Data/Property[@Name="PropertyName"]

For example, a script creating performance data might create a property bag with values in the following table. This table shows the name of the value created by the script and the corresponding $Data variable that would be used to map the property bag data to performance data.

|  |  |  |
| --- | --- | --- |
| Property Bag Value Name | Sample Value | Variable |
| ObjectName | MyObject | $Data/Property[@Name='ObjectName']$ |
| CounterName | MyCounter | $Data/Property[@Name='CounterName']$ |
| InstanceName | MyInstance | $Data/Property[@Name='InstanceName']$ |
| Value | 10 | $Data/Property[@Name='Value']$ |

### Typed Property Bags

Values from a property bag created by using the CreatePropertyBag method may be used by the workflow to create any type of data including discovery data, performance data, and events. The CreateTypedPropertyBag method can be used to create property bags that may only be used for a particular data type. The resulting property bag is identical and is referred to with the same $Data variables.

The CreateTypedPropertyBag functions to ensure that the output of a script is only used for the specific data type that it was intended. It is typically not used in scripts created by using a wizard in the Authoring Console but instead by scripts included in custom modules that may be used in multiple workflows. This concept is discussed in the [Composition](#z94c023257fa84110b5234f4b8b2f927c) section of this guide.

The syntax of the CreateTypedPropertyBag method is as follows:

MOMScriptAPI.CreateTypedPropertyBag(type)

The parameter type specifies the type of data that may be created from values in the property bag. Possible values are shown in the following table:

|  |  |
| --- | --- |
| Value | Data Type |
| 0 | Alert |
| 1 | Event |
| 2 | Performance |
| 3 | State |

## Basic Script Structure

The following sample shows a monitoring script with the following characteristics.

 Accepts arguments for the name of the computer that is running the script and a path of the location of the application.

 Creates a property bag with the values named ComputerName, InstanceName, and PerfValue.

sComputerName = WScript.Arguments(0)

sApplicationPath = WScript.Arguments(1)

Set oAPI = CreateObject("MOM.ScriptAPI")

Set oBag = oAPI.CreatePropertyBag()

oBag.AddValue "ComputerName", sComputerName

oBag.AddValue "InstanceName", "MyInstance"

oBag.AddValue "Value", 1.0

oAPI.Return(oBag)

## Script Breakdown

Details of each section of the script are discussed here.

sComputerName = WScript.Arguments(0)

sApplicationPath = WScript.Arguments(1)

The first two lines of the script accept arguments. These values would be expected to be in the Arguments parameter of the rule or monitor running the script. The script can use any number of arguments that are required for the logic of the script.

Set oAPI = CreateObject("MOM.ScriptAPI")

Set oBag = oAPI.CreatePropertyBag()

The next two lines create a property bag. These lines will also be unchanged in most monitoring scripts. The main purpose of the rest of the script will be to add values to the property bag by using data that is collected from the agent computer.

oBag.AddValue "ComputerName", sComputerName

oBag.AddValue "InstanceName", "MyInstance"

oBag.AddValue "Value", 1.0

After the property bag is created, any number of values can be added to it. You do this with the AddValue method on the property bag object by using the name of the item followed its value. This example uses explicit values. In actual monitoring script, additional code would be expected that would collect information from the agent computer to include in these values.

oAPI.Return(oBag)

After all values are added to the property bag, it is returned into the workflow. This line is required, and without it the property bag is discarded when the script ends. This method is only used when the script creates only a single property bag. For more information about scripts that return multiple property bags and conditions when such a strategy is used, refer to the [Cookdown](#z1f8fb91b1e814c8daacf758edb2f3141) section of this guide.

## Defining a Script

### Script Name

Scripts in a management pack must be given a name with either a .vbs or ,js extension depending on their language. This name is used to create the file in the temporary directory on the agent. There is no requirement to make this name unique because each script is provided its own temporary directory on the agent.

### Script Arguments

Arguments are provided to the script in a single line separated by spaces. This is identical to the command line that would be provided if the script were run on a command line. This is access in the Authoring console from the Parameters button.

Each argument can be either an explicit value or a $Target variable to use the value of a property on the target object. Any $Target variables are resolved when the script is run so that the script is provided with the resolved values on the command line.

Important

Any $Target variable that might resolve to a value that includes a space should be enclosed with quotation marks. If a value includes spaces and does not have quotation marks, then it will be seen by the script as two separate arguments. The quotation marks will ensure that the value is seen as a single argument.

For example, the sample script earlier expects two arguments for the computer name and the application path. Assuming this was part of a workflow targeted at a class hosted by the Windows Computer class, the computer name could be retrieved from the PrincipalName property. If the application path were a property on the target class, then the arguments might look similar to the following example. Notice the quotation marks around the ApplicationPath property, because this could resolve to a value that contains a space.

$Target/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/PrincipalName$ "$Target/Property[Type="MyApp.MyClass"]/ApplicationPath$"

### Script Timeout

All scripts are given a timeout value which indicates the number of seconds that the script can run before the agent stops it. This prevents problem scripts from running continuously and putting excess overhead on the agent computer.

The timeout value assigned to a script should allow enough time for the script to run under ordinary conditions, but should be less than the interval that the script is scheduled to run. If a script is configured to have a timeout value greater than its duration, then possibly multiple copies of the script could be running concurrently.

## Windows PowerShell

The following script is the Windows PowerShell equivalent of the VBScript sample monitoring script shown previously. Both scripts use the same MOM.ScriptAPI object and methods.

param($computerName,$version)

$api = New-Object -comObject 'MOM.ScriptAPI'

$bag = $api.CreatePropertyBag()

$bag.AddValue 'ComputerName', $computerName

$bag.AddValue 'InstanceName', '1.0'

$bag.AddValue 'Value', '1.0'

$bag

Other than obvious syntax, there are two primary differences between the VBScript and Windows PowerShell scripts, as follows.

param($computerName,$version)

The first difference is in the means of retrieving script arguments. Windows PowerShell can accept positional arguments as VBScript does, but named parameters that use the param command better support the Windows PowerShell modules included with Operations Manager 2007 R2. These modules let parameters be separately specified in the module configuration where they are retrieved by the script as named parameters.

$bag

Another important difference is the way that the property bag data is returned to the workflow. In VBScript, the Return method of the MOM.ScriptAPI object is used. In Windows PowerShell, this method will not return the data correctly. The Return method sends the discovery data to the Standard Out (StdOut) stream, whereas Windows PowerShell requires the data to be sent to the output pipeline. This is achieved by typing the property bag variable on its own line.

Monitors and rules that use PowerShell scripts cannot be created by using wizards in the Authoring Console. Custom workflows must be created by using the PowerShell modules available in Operations Manager 2007 R2. Information on custom workflows is provided in the [Composition](#z94c023257fa84110b5234f4b8b2f927c) section of this guide.

For management packs that must support Operations Manager 2007 SP1, the PowerShell modules cannot be used. In this case, a custom solution running PowerShell.exe from a custom workflow would have to be employed.

## Supported Workflows

Scripts can be used in the following kinds of workflows:

 Monitors

 Collection Rules

Collection rules based on scripts can map values in the property bag to either events or performance data. There is no difference in the scripts for either of these kinds of rules. The only difference is in the properties of the resulting data type that the property bag values are mapped into.

## See Also

[Script Monitors](#zc5fdd47a651c4c488f606e54ca9c0e3c)

# Monitors and Health State

Every instance of each class in a System Center Operations Manager 2007 service model has a health state that represents the current health of the object. This health state is set by one or more monitors targeted at the class. The health state of each monitor changes as the health of the components of the application represented by the class change.

There are three kinds of monitors as shown in the following table:

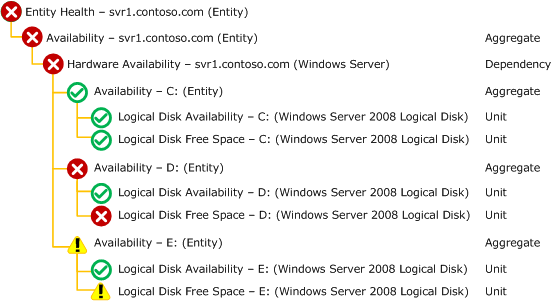
|  |  |
| --- | --- |
| Monitor | Description |
| [Unit Monitors](#z9130c17a58234046a7c719740c0d0ab9) | Measures some aspect of the application. This might be checking a performance counter to determine the performance of the application, running a script to perform a synthetic transaction, or watch for an event that indicates an error. Classes will typically have multiple unit monitors targeted at them to test different features of the application and to monitor for different problems. |
| [Dependency Monitors](#zfee2d08685ed40439fe1f8df64226740) | Provides health rollup between different classes. This allows the health of an object to depend on the health of another kind of object that it relies on for successful operation. |
| [Aggregate Monitors](#z8ecc81f8ad744a58a95b1195472662b1) | Provides a combined health state for similar monitors. Unit and dependency monitors will typically be configured under a particular aggregate monitor. In addition to providing better general organization of the many different monitors targeted at a particular class, aggregate monitors provide a unique health state for different categories of the class. |

Monitors each have either two or three states. At any time, a monitor will be in one and only one its potential states. When a monitor loaded by the agent, it is initialized to a healthy state. The state will change only if the specified conditions for another state are detected.

The overall health of a particular object is determined from the health of each of its monitors. This will be a combination of monitors targeted directly at the object, monitors target at objects rolling up to the object through a dependency monitor, dependency monitors targeted at those objects, and so on. This hierarchy is illustrated in the Health Explorer of the Operations console. The policy for how health is rolled up is part of the configuration of the aggregate and dependency monitors.

The following diagram shows an example of the Health Explorer for the Windows Server class. This shows the use of the different kinds of monitors contributing to an overall health state.

Sample Health Explorer



## In This Section

[Unit Monitors](#z9130c17a58234046a7c719740c0d0ab9)

|  |
| --- |
| Details on unit monitors using different data sources. |

[Dependency Monitors](#zfee2d08685ed40439fe1f8df64226740)

|  |
| --- |
| Details on dependency monitors. |

[Aggregate Monitors](#z8ecc81f8ad744a58a95b1195472662b1)

|  |
| --- |
| Details on aggregate monitors. |

[Alerts from Monitors](#zc0732e8daf914b38849278fa39d31051)

|  |
| --- |
| Details on alerts that are created from monitors and using variables in alert descriptions. |

[Diagnostics and Recoveries](#z6d472fd061a64c7ca44f19ddb8e3cd6e)

|  |
| --- |
| Details on diagnostics and recoveries that run in response to the change in a monitor’s health state. |

# Unit Monitors

Unit monitors measure some aspect of the health of a particular class. The may detect an event that indicate a problem, sample a performance counter for comparison against a threshold, or run a synthetic transaction to test some feature of the application.

## In This Section

Unit monitors are primarily defined by data source and the logic that is used with this data to determine whether the state should be changed. The following sections provide details on each of the different kinds of unit monitor and the data sources and logic that are available for each.

[Event Monitors](#z9a556512ab434d75aa4dd6958c0f3457)

|  |
| --- |
| Monitors by using events to identify issues affecting application health. |

[Performance Monitors](#z7335542b6b38488aa703e6eaae902a46)

|  |
| --- |
| Monitors by using values of performance counters compared to threshold values to determine a health state. |

[Script Monitors](#zc5fdd47a651c4c488f606e54ca9c0e3c)

|  |
| --- |
| Monitors by running a scheduled script to collect health information. |

[Service Monitors](#zcf5c92c4ee4a47cdbc24a6a1d51d6b93)

|  |
| --- |
| Monitors checking the running state of a service. |

# Event Monitors

Event monitors use one of the event data sources to identify a particular event that indicates an issue. As soon as the specific data source that holds the required information is identified, the logic used to determine different health states must be determined. In addition to the logic that indicates whether an error condition has occurred, additional logic must be defined to determine when the state should be changed back to a healthy condition.

## Detection Logic

The different kinds of logic that can be used to detect an error condition by using events are listed in the following table. As noted in the table, some logic can only be used with Windows events.

|  |  |  |
| --- | --- | --- |
| Logic | Data Sources | Description |
| Simple Event | All | Detects an error state from the occurrence of a single event. |
| Repeated Events | All | Detects an error state from one or more occurrences of a particular event in a specified time window. |
| Correlated Events | Windows Events | Detects an error state from the occurrence of two events in a specified time window. |
| Correlated Missing Events | Windows Events | Detects an error state from an expected event not being detected in a particular time window after the occurrence of another event. |
| Missing Event | Windows Events | Detects an error state from an expected event not being detected in a particular time window. |

### Simple Event

Simple detection refers to a state change being triggered immediately after a single occurrence of the specified event. This is the most basic kind of detection and will apply to most scenarios.

### Repeated Events

Repeated event detection uses one or more occurrences of a particular event in a time window to indicate an error condition. This typically applies to conditions in an application where a single event on its own can be ignored, but multiple occurrences of that event in a particular time window indicate a potential error. There are different algorithms that can be used for this detection, depending on the logic that best identifies the specific application issue. The following are details of the different algorithms:

#### Trigger on Timer

Trigger on timer consolidation of events uses a specified time window and is not dependent on the number of events received. A single event can trigger an error in the health state as in simple detection. Unlike simple detection which sets the health state immediately upon detection of the specified event, however trigger on timer consolidation waits until a specified time window to set the health state of the monitor. The time window can be a rotating time duration of specified length or a specific window based on day of the week.

Trigger on timer consolidation is useful for errors that should only be detected in a certain time window. Used with a time window based on a specific time of day, this disables the monitor outside that time period. It can also have the effect of delaying the change of state for a particular time during which an event that indicates a healthy state could be received. In this case, the health state would never be changed.

#### Trigger on Count

Trigger on count consolidation of events lets a monitor require multiple occurrences of the same event in a specified time window before it changes the health state to an error. The time window can be rotating time duration of specified length or a specific window based on day of the week.

Trigger on count consolidation resembles trigger on timer consolidation except that multiple occurrences of the event are required instead of just one. When the time window is reached, the event count is returned to zero, and the specific number of events must detected before the time window expires again for the health state to be changed.

#### Trigger on Count, Sliding

Trigger on count, sliding consolidation of events is similar to trigger on count consolidation except that the time window is reset every time that the specified event is received. The time window only expires if the time is reached after the occurrence of the last event.

Trigger on count, sliding consolidation is useful for error conditions that are detected by a certain number of events in a particular length of time. By using trigger on count consolidation, some events could be received in one time window and then other events received in the next time window with the result that the health state is never changed. Using trigger on count, sliding consolidation, the time window depends on when the event occurs preventing this condition.

#### Repeated Events Example

To help with understanding the different algorithms used for repeated event detection, the following table shows the effect on health state for monitors based on the different kinds of consolidation. This is based on a repeated event monitor that uses the following details:

 Consolidation interval: 2 minutes

 Compare count: 3 (ignored by Trigger on Timer)

 Health state on repeated event: Critical

 Reset Logic: Event reset using Event 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time | Event | Trigger on Timer | Trigger on Count | Trigger on Count, Sliding |
| 00:00:00 | - | Healthy | Healthy | Healthy |
| 00:01:00 | Event 1 | Healthy | Healthy | Healthy |
| 00:02:00 | - | Healthy | Healthy | Healthy |
| 00:02:30 | - | Healthy | Healthy | Healthy |
| 00:03:00 | - | Critical | Healthy | Healthy |
| 00:03:30 | Event 3 | Healthy | Healthy | Healthy |
| 00:04:00 | Event 1 | Healthy | Healthy | Healthy |
| 00:04:30 | - | Healthy | Healthy | Healthy |
| 00:05:00 | Event 1 | Critical | Healthy | Healthy |
| 00:05:30 | - | Critical | Healthy | Healthy |
| 00:06:00 | - | Critical | Healthy | Healthy |
| 06:30:00 | Event 1 | Critical | Healthy | Healthy |
| 07:00:00 | Event 1 | Critical | Healthy | Critical |
| 07:30:00 | - | Critical | Healthy | Critical |
| 00:08:00 | Event 1 | Critical | Healthy | Critical |
| 00:08:30 | - | Critical | Critical | Critical |
| 00:09:00 | Healthy | Critical | Healthy | Healthy |

 Using trigger on timer, a critical state is set at 00:03:00 event though the event is received at 00:01:00 because the time window starts when the monitor is loaded. The start is reset to healthy at 00:03:30, but the critical state is again triggered at 00:05:00 from the time window started at 00:03:00.

 Using trigger on count, the event at 00:05:00 does not trigger a critical state because the time window started by the event at 00:01:00 would have expired at 00:03:00. This event is instead part of the time window started by the event at 00:04:00 which expires at 00:06:00. The monitor triggers a critical state at 00:08:30 because of the 3 events detected in the time window started with the event at 00:06:30.

 Using trigger on count, sliding, each occurrence of Event 1 starts its own window. The critical state is triggered at 00:07:00 from the 3 events detected in the time window started with the event at 00:05:00.

#### Properties

Repeated event monitors have the following additional properties that may be used in criteria and message descriptions:

|  |  |
| --- | --- |
| Property | Description |
| TimeWindowStarts | Time that the time window started |
| TimeWindowEnds | Time that the time window ended |
| TimeFirst | Time of the first event |
| TimeLast | Time of the last event |
| Count | Number of events that are collected in the time window |

### Correlated Events

A correlated event monitor uses two separate events in a particular time period to detect a single issue. This kind of monitor supports conditions where an issue cannot be identified by a single event alone.

When the first event is detected, a timer is triggered. If the second event is received within that period, the state change is triggered. If the second event is not received in the period, the timer is reset until the first event is received again. The monitor may be configured to better tune the specific conditions that must be met in order to perform correlation. These options include the following:

 Whether the events must be in chronological order. One of the events may always be expected before the other one, or they may be expected in either order.

 Whether the first or last occurrence of the first event should be used. If the first occurrence is specified, then each occurrence of the first event will have its own time window and search for corresponding occurrences of the second event. With the last occurrence specified, if the first event reoccurs with the time window, then the time window is extended based on the last event. The monitor can also be configured to reset the time window every time that the first event occurs. When the time window is reset, all previous occurrences of both events are ignored.

 The number of occurrences of the second event that must be received to trigger the state change. Instead of changing the health state after receiving a single instance of the two events, multiple instances of the second event may be required.

 Properties between the first and second event that must match for correlation to be performed. Instead of detecting two occurrences of each event, additional comparison may be required to determine whether the events are related. The monitor can, for example, confirm that a particular parameter matches between the two events to make sure that they match.

#### Correlated Events Example

The following table provides an example of a correlated event monitor by using the first and the last occurrence of the first event. The monitor uses the following details:

 Event Log A: Event 1

 Event Log B: Event 2

 Correlation interval: 2 minutes

 Number of occurrences of Event 2: 3

 Health state on correlation: Critical

 Reset Logic: Event reset using Event 3

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Event | First Occurrence | Last Occurrence |
| 00:00:00 | - | Healthy | Healthy |
| 00:01:00 | Event 1 | Healthy | Healthy |
| 01:30 | Event 2 | Healthy | Healthy |
| 00:02:00 | Event 2 | Healthy | Healthy |
| 00:02:30 | - | Healthy | Healthy |
| 00:03:00 | Event 1 | Healthy | Healthy |
| 00:03:30 | Event 2 | Healthy | Healthy |
| 00:04:00 | Event 2 | Healthy | Healthy |
| 00:04:30 | Event 1 | Healthy | Healthy |
| 00:05:00 | Event 2 | Critical | Healthy |
| 05:30:00 | Event 3 | Healthy | Healthy |
| 06:00:00 | Event 1 | Healthy | Healthy |
| 06:30:00 | Event 2 | Healthy | Healthy |
| 07:00:00 | Event 1 | Healthy | Healthy |
| 07:30:00 | Event 2 | Healthy | Healthy |
| 08:00:00 | Event 2 | Critical | Healthy |
| 08:30:00 | Event 2 | Critical | Critical |
| 09:00:00 | Event 3 | Healthy | Healthy |

 The First Occurrence does not trigger a critical state when Event 2 is detected at 00:03:00 because the timer was reset at 00:03:00 which is 2 minutes after the first occurrence of Event 1 at 00:01:00.

 The First Occurrence triggers a critical state at 00:05:00 because Event 2 is detected 3 times within the 2 minutes since the first occurrence of Event 1 at 00:03:00. Event 1 starts a new time window at 00:03:00 because the time window from Event 1 at 00:01:00 would have expired.

 The First Occurrence triggers a critical state at 00:08:00 because Event 2 is detected 3 times within 2 minutes from Event 1 at 00:06:00.

 The First Occurrence resets its state to healthy at 00:05:30 and 00:09:00 because Event 3 is detected.

### Correlated Missing Events

A correlated missing event monitor determines an error by the absence of a particular event after the occurrence of another. This resembles the missing event monitor except that instead of searching for the missing event in a particular time window, the monitor searches for the event in a particular time after another event is first detected.

For example, consider an application that performs a backup each evening and creates an event when it starts and a second event when it has completed successfully. A correlated missing event monitor could be created that searches for the event in a particular time window each evening. If both events are detected, then the monitor remains in a healthy state. If the first is found, then the timer starts. If the time is reached before the second event is detected, then the state change is triggered to indicate that the last backup did not occur successfully.

#### Correlated Missing Events Example

The following table provides an example of a correlated missing event monitor by using the first and the last occurrence of the first event. The monitor uses the following details:

 Missing Event Log A: Event 1

 Missing Event Log B: Event 2

 Correlation interval: 2 minutes

 Number of occurrences of Event 2: 3

 Health state on correlation: Critical

 Reset Logic: Event reset using Event 3

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Event | First Occurrence | Last Occurrence |
| 00:00:00 | - | Healthy | Healthy |
| 00:01:00 | Event 1 | Healthy | Healthy |
| 1:30 | Event 2 | Healthy | Healthy |
| 00:02:00 | Event 2 | Healthy | Healthy |
| 00:02:30 | Event 1 | Healthy | Healthy |
| 00:03:00 | - | Critical | Healthy |
| 00:03:30 | Event 2 | Critical | Healthy |
| 00:04:00 | Event 2 | Critical | Healthy |
| 00:04:30 | - | Critical | Critical |
| 00:05:00 | Event 3 | Healthy | Healthy |

 The First Occurrence triggers a critical state at 00:03:00 because Event 2 has not been detected 3 times in the 2 minute interval since the first occurrence of Event 1 at 00:01:00.

 The Last Occurrence does not trigger a critical state at 00:03:00 because Event 1 occurs at 00:02:30 resetting the timer. The critical state is not triggered until 00:04:30 when Event 2 has not been detected in the 2 minutes interval since the last occurrence of Event 1 at 00:02:30.

 The single occurrence of Event 3 at 00:05:00 resets both monitors to healthy.

### Missing Event

Instead of detecting a particular event to identify an error condition, a missing event monitor uses the absence of a particular event in a particular time window to determine an error. This supports applications that are expected to generate an informational event that indicates a successful operation or the success of a particular action.

For example, consider an application that performs a scheduled data transfer each evening and creates an event when it has completed successfully. A missing event monitor could be created that searches for the event in a particular time window each evening. If the event is detected, then the monitor remains in a healthy state. If it is not found, then it enters error state that indicates that the last transfer did not occur successfully.

#### Missing Event Example

The following table provides an example of a missing event monitor by using the following details:

 Event: Event 1

 Fixed Schedule: Su-Sa 2:00 AM – 3:00 AM

 Health state on missing event: Critical

 Reset Logic: Event reset using Event 3

|  |  |  |
| --- | --- | --- |
| Time | Event | Health State |
| 00:00:00 | - | Healthy |
| 00:01:00 | Event 1 | Healthy |
| 00:02:00 | - | Healthy |
| 00:03:00 | - | Healthy |
| 00:04:00 | - | Healthy |
| 00:05:00 | Event 3 | Healthy |

 The critical state is triggered at 00:03:00 when Event 1 is not detected within the specified window.

## Health Reset Logic

The previous detection criteria describe the conditions under which a monitor changes to a warning or critical state. In addition to detecting an error state, each monitor must have logic defined to determine when the state should be returned to healthy. The different methods for resetting state are shown in the following table:

|  |  |
| --- | --- |
| Reset Logic | Description |
| Event Reset | A single specific event indicates that monitor should be reset. |
| Manual Reset | The monitor is never automatically rest. The user must manually reset the monitor. |
| Timer Reset | The monitor is automatically reset after a specified time. |

Each of these methods is discussed at length in the following sections:

### Event Reset

With event reset, the monitor is reset when a single occurrence of a specific event is detected. The event must be the same type as the event used for detecting the error condition. For example, a Windows event monitor might specify an event with a particular event source and number to indicate an error condition. Another Windows event with the same event source but a different number might indicate that the error in the application was corrected.

Event reset can only be used if the application provides an event indicating the particular error was corrected. Many applications create an event when an error occurs but may not create a corresponding event that indicates that the error was corrected. Event reset cannot be used in this case.

### Manual reset

With manual reset, the monitor never returns to a healthy state automatically. The user must determine whether the problem was corrected and then select the monitor in the Health Explorer and select Reset Health.

The advantage to this strategy is that a monitor can be used for issues that do not create an event that indicates a healthy state. The monitor can affect the health state of the managed object instead of creating a simple alert from a rule. The downtime will be recorded for the object in the State Change Events in the Operations Console and in any availability reports.

There are multiple implications of this strategy that should be considered. The first is the additional work required from the user because the monitor will never automatically reset. It can also result in too much downtime being recorded if the user waits a long time before performing the reset. The problem may have been corrected fairly quickly, but the healthy state will not be recorded until the user performs the reset.

Use of manual reset should be especially cautioned for monitors where there is a potential for a single problem to affect multiple instances of the target class. Because users cannot reset the monitor for multiple instances in the Operations Console, the user would be required to manually open the Health Explorer for each instance to perform this action. Depending on the number of instances, this could result in significant effort for the user.

### Timer Reset

A timer reset acts the same as a manual reset except that if the user does not manually reset the monitor after a specified time, it will reset automatically. One use of this kind of reset is for issues that continuously log error events until the problem is corrected. Instead of using another event to indicate that the problem was corrected, the previously detected error event for a specified period can be used as the success criteria.

The timer reset can be used in the place of a manual reset providing the advantage of automatically resetting after a while if the user does not perform a manual reset.

# Performance Monitors

Monitors in a System Center Operations Manager 2007 management pack based on performance counters collect numeric data at set intervals and compare it to one or more threshold values. This may be a simple comparison that compares each sample to a single threshold or more complex logic, depending on the requirements of the application.

## Threshold Types

Multiple kinds of calculations may be performed to determine the threshold for a performance monitor. These threshold types are listed in the following table:

|  |  |  |
| --- | --- | --- |
| Threshold Type | Number of States | Description |
| Average Threshold | 2 | Compare the average of multiple collected values to a threshold. |
| Consecutive Samples | 2 | Compare several consecutive values to a threshold. All collected values must match the threshold criteria. |
| Delta Threshold | 2 | Compare the change between two consecutive values to a threshold. |
| Double Threshold | 3 | Compare a single collected value to two thresholds with one that indicates a Warning state and the other that indicates a Critical state. |
| Simple Threshold | 2 | Compare a single collected value to a threshold. |

Each kind of logic is described in detail in the following sections:

### Simple Threshold

The simple threshold type is the most basic kind of performance threshold. A single numeric value is provided for the threshold. This threshold is compared to the measured value of the performance data.

Simple threshold supports a two state monitor. One state is set by a performance value equal to or less than the threshold. The other state is set by a performance value greater than the threshold.

### Double Threshold

The double threshold type is similar to the simple threshold type but allows for two thresholds to be specified. Each threshold is compared to the measured value of the performance data.

Double threshold supports a three state monitor. One state is set by a performance value less than the low threshold. Another state is set by a performance value that is greater than or equal to the low threshold or one that is less than or equal to the high threshold. Another state is set by a value that is greater than the high threshold.

The following table provides an example of a double monitor by using the following details:

 Sample rate: 5 minutes

 Low threshold value: 10

 High threshold value: 15

 Over Upper Threshold State: Critical

 Between Thresholds State: Warning

 Under Lower Threshold State: Healthy

|  |  |  |
| --- | --- | --- |
| Time | Value | State |
| 00:00:00 | 5 | Healthy |
| 00:05:00 | 10 | Warning |
| 00:10:00 | 12 | Warning |
| 00:15:00 | 9 | Healthy |
| 00:20:00 | 12 | Warning |
| 00:25:00 | 16 | Critical |
| 00:30:00 | 15 | Critical |
| 00:35:00 | 8 | Healthy |

 The warning threshold is first exceeded at 00:05:00, but the value does not exceed the critical threshold.

 The critical threshold is first exceeded at 00:25:00 when the state is changed from warning to critical.

 The state is returned to a healthy state at 00:15:00 and 00:35:00 when the sampled value is less than the warning threshold.

### Average Threshold

The average threshold type calculates the average of a specified number of consecutive samples and compares it to the specified threshold.

Average threshold supports a two state monitor. One state is set by an average performance value equal to or less than the threshold. The other state is set by an average performance value greater than the threshold.

The following table provides an example of an average threshold monitor by using the following details:

 Sample rate: 5 minutes

 Threshold value: 10

 Number of samples: 3

 Over Threshold State: Critical

 Under Threshold State: Healthy

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Value | Average | State |
| 00:00:00 | 5 | - | Healthy |
| 00:05:00 | 10 | - | Healthy |
| 00:10:00 | 12 | 9.0 | Healthy |
| 00:15:00 | 9 | 10.3 | Critical |
| 00:20:00 | 12 | 11.0 | Critical |
| 00:25:00 | 14 | 11.7 | Critical |
| 00:30:00 | 11 | 12.3 | Critical |
| 00:35:00 | 4 | 9.7 | Healthy |

 Because the specified number of samples for the average calculation is 3, no value is evaluated until the third sample.

 The value of 12 sampled at 00:10:00 exceeds the threshold value, but the calculated average from the last 3 samples is 9.0, which is under the threshold. The state is not changed.

 The value of 9 sampled at 00:15:00 does not exceed the threshold. But the calculated average from the last 3 samples is 10.3 which does exceed the threshold. The state is changed.

 The monitor does not return to a healthy state until 00:35:00 when the average from the last 3 samples drops the under the threshold value.

### Consecutive Samples

The consecutive threshold type compares the threshold value to the performance counter for several consecutive samples. This supports monitors that should not be triggered by only a single value exceeding a threshold. The threshold must be exceeded multiple consecutive times to trigger a change in state.

Consecutive threshold supports a two state monitor. One state is set by the value being either greater than or less than the threshold value for each consecutive sample. The other state is set by a single sample not matching the other criteria.

The following table provides an example of a consecutive sample monitor by using the following details:

 Sample rate: 5 minutes

 Threshold value: greater than or equal to 10

 Number of samples: 3

 Over Threshold State: Critical

 Under Threshold State: Healthy

|  |  |  |
| --- | --- | --- |
| Time | Value | State |
| 00:00:00 | 5 | Healthy |
| 00:05:00 | 10 | Healthy |
| 00:10:00 | 12 | Healthy |
| 00:15:00 | 9 | Healthy |
| 00:20:00 | 12 | Healthy |
| 00:25:00 | 14 | Healthy |
| 00:30:00 | 11 | Critical |
| 00:35:00 | 8 | Healthy |

 The threshold is exceeded by the values sampled at 00:05:00 and 00:10:00, but the value at 00:15:00 is under threshold and resets the count.

 The value at 0:30:00 is the first time that 3 consecutive values have been sampled that exceed the threshold, so the state is changed.

 The single value at 00:35:00 is under the threshold and resets the monitor to a healthy state.

### Delta Threshold

The delta threshold type compares the threshold value to the difference between two performance values. This might be two consecutive values or two values separated by a specified number of samples.

Delta threshold supports a two state monitor. One state is set by the difference of two values being greater than the threshold value. The other state is set by the difference of two samples being equal to or less than the threshold value.

The following table provides an example of a delta threshold monitor by using the following details:

 Sample rate: 5 minutes

 Threshold value: 10

 Number of samples: 3

 Over Threshold State: Critical

 Under Threshold State: Healthy

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Value | Delta |  |
| 00:00:00 | 7 | - | Healthy |
| 00:05:00 | 8 | - | Healthy |
| 00:10:00 | 13 | - | Healthy |
| 00:15:00 | 16 | 9 | Healthy |
| 00:20:00 | 21 | 13 | Critical |
| 00:25:00 | 24 | 11 | Critical |
| 00:30:00 | 25 | 9 | Healthy |

 Because the specified number of samples that the delta should be calculated from the current sampled value to the value 3 samples behind, no value is evaluated until the fourth sample.

 The delta calculation exceeds the threshold value at 00:20:00, and the state is changed.

 The monitor is reset at 00:30:00 when the delta calculation falls under the threshold.

## Self-tuning Threshold Monitors

A self-tuning threshold monitor uses a learning process to determine the typical values for a specified performance counter object and automatically sets the threshold levels based on the learned values. Avoid self-tuning threshold monitors because they may not work well in most customer environments.

# Script Monitors

Script monitors run a monitoring script regularly and evaluate the results to determine the state of the monitor. The script could perform such actions as running a synthetic transaction against an application, gathering performance data to be evaluated against a threshold, or retrieving a status of some aspect of the application. Script monitors incur more overhead than the other types of monitors and should be used only when one of those monitors does not provide the required functionality.

Script monitors can use either two states or three states. Criteria must be defined for each state using values from the property bag created by the script. The kinds of values in the property bag will vary depending on the particular script. A numeric value might be compared to a threshold value as in a performance monitor. In that case, the healthy state might be defined by the value being under the threshold value while the critical state is defined by the value being over the same threshold. A synthetic transaction might return a text result indicating whether the test was successful or not. In that case, the criteria for each state would be the string indicating that particular health.

## See Also

[Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8)

[How to create a monitor based on a script](#zfac2ebbdd83c4c18b2e7c567a2269d2b)

[How to create a monitor based on a Windows PowerShell script](#z64c66f4abce24bb793c449de3be6035b)

# Service Monitors

Service monitors measure the running state of a Windows service. There is no configuration required other than the name of the service. This is a two state monitor with the monitor sets the monitor to a healthy state if the service is running and a critical state if the service is not running. The monitor can be configured to check the startup type of the service. This ensures that the service is only monitored if its startup type is set to automatic.

# Aggregate Monitors

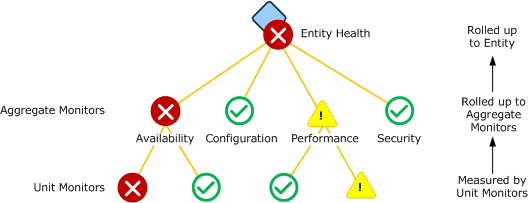
Aggregate monitors group multiple monitors to provide a single health aggregated health state. This provides an organization to all of the monitors targeted at a particular class and provides a consolidated health state for specific categories of operation.

## Standard Aggregate Monitors

Every class has four standard aggregate monitors: Availability, Configuration, Performance, and Security. These are in the System.Health.Library management pack and targeted at the Entity class. Because all classes inherit from the Entity class, all classes inherit these standard monitors. The standard set of aggregate monitors will be sufficient for most classes.

Most monitors will fall into one of the four categories represented by the standard aggregate monitors. Because of this, custom aggregate monitors will typically use one of the standard aggregate monitors as their parent instead of being positioned alongside them directly under the entity health. Unit monitors and dependency monitors will similarly use either a custom aggregate monitor or one of the standard aggregate monitors as their parent.

Standard aggregate monitors



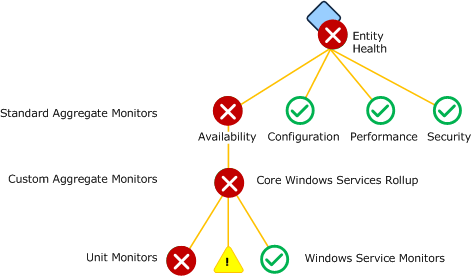
## Custom Aggregate Monitors

Management packs can include custom aggregate monitors specific to the requirements of classes in a particular application. These monitors may use another aggregate monitor for their parent or the top level Entity State similar to the standard aggregate monitors use. Custom aggregate monitors can be configured underneath another aggregate monitor or attached directly to the entity state.

For example, the Windows Server 2008 Operating System (Monitoring) management pack includes an aggregate monitor called Microsoft.Windows.Server.2008.OperatingSystem.CoreServicesRollup that is used to combine the health of the different services that are monitored by this management pack. There are nine services that the management pack considers critical to the operation of a computer running Windows Server 2008. Instead of positioning these directly under the Availability aggregate monitor alongside other unit monitors, the aggregate monitor provides a combined health measurement for all the related services.

This aggregate monitor is illustrated in the following diagram.

Core Windows Services Rollup aggregate monitor



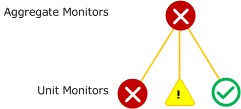
## Health Rollup Policy

Each aggregate monitor must define a health rollup policy which is the logic that is used to determine the health of the aggregate monitor based on the health of the monitors under it. The possible health rollup policies for an aggregate monitor are as follows:

### Worst state

The state of the aggregate monitor matches the state of the child monitor with the worst health state. This is the most common policy used by aggregate monitors.

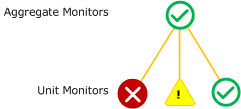
Worst state health policy



### Best state

The state of the aggregate monitor matches the state of the child monitor with the best health state.

Best state health policy



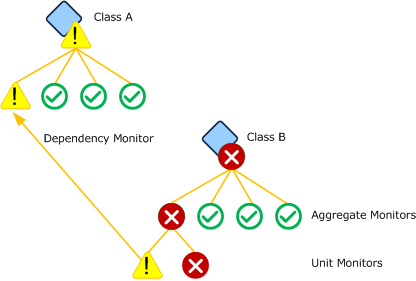
# Dependency Monitors

Dependency monitors let the health of one object be affected by the health of another object. This allows for health rollup between specific related instances of different classes.

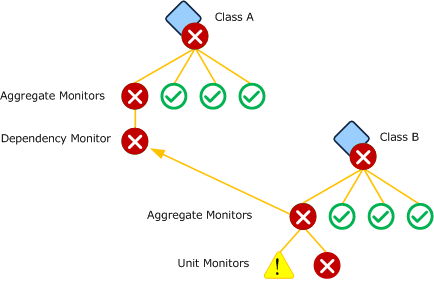
Each dependency monitor is based on a specific hosting or containment relationship. Just creating a relationship between two objects does not alone provide rollup between their health states. A dependency monitor must be associated with the relationship for rollup of health to be performed.

The source and target class for a dependency monitor are defined by the relationship that the monitor is based on. The monitor must additionally specify a specific unit monitor or aggregate monitor on the target class and an aggregate monitor on the source class. Only the health of the target monitor is considered when calculating the health of the dependency monitor, and it only affects the health of the specified aggregate monitor on the target object.

Dependency monitor based on unit monitor

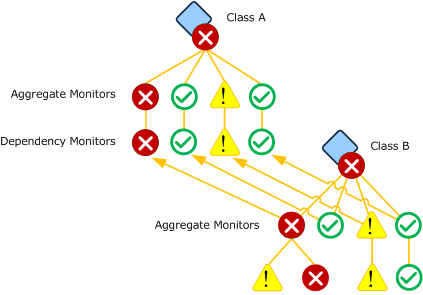


Dependency monitor based on aggregate monitor



Multiple dependency monitors can be created on a single relationship if the health of the source class should be affected by multiple unit or aggregate monitors on the target class. For example, a dependency monitor might be created for each standard aggregate monitor as shown in the following image.

Multiple dependency monitors for a single class



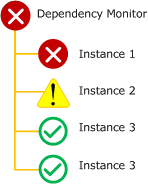
## Health Rollup Policy

There may be multiple instances of the target class, each with a different health state. Each dependency monitor must define a health rollup policy to define the logic that is used to determine the health of the dependency monitor based on the health of the instances of its target monitor. The possible health rollup policies for a dependency monitor are as follows:

### Worst state policy

The source object matches the state of the target object that has the worst health state. This is used when the source object should only be healthy if all the target objects are healthy. This is the most common policy used by dependency monitors.

Worst state health policy

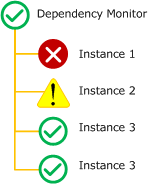


### Best state policy

The source object matches the state of the target object that has the best health state. This policy is used when only one of the source objects has to be healthy for the target object to be healthy.

For example, the Microsoft Windows Hyper-V 2008 Monitoring management pack has a dependency monitor on the hosting relationship from Microsoft.Windows.HyperV.ServerRole to Microsoft.Windows.HyperV.VirtualNetwork that uses a best state policy. This is because the Hyper-V server is functional as long as it has one functional virtual network. The logic defined by this management pack is that the server class should show an error state if no virtual networks are available.

Best state health policy

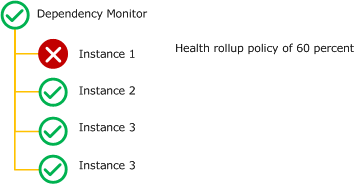


### Percentage policy

The source object matches the worst state of a single member of a specified percentage of target objects in the best state. This policy is used when a certain percentage of target objects must be healthy for the target object to be considered healthy.

For example an application might run on a web farm that includes multiple Web servers. Because of the redundancy offered in this kind of deployment, the application might be considered healthy if a particular percentage of servers is available. The farm itself could be represented in the management pack by a health rollup class based on System.ApplicationComponent with a containment relationship to the Web servers. A dependency monitor could be created on this containment relationship with a health rollup policy specifying a percentage. Even if one or more Web servers had a problem, as long as the specified percentage were in a health state then the class representing the web farm would also be healthy.

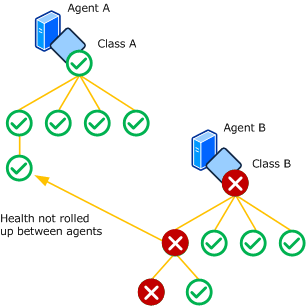
Percentage health policy



## Health Rollup between Agents

Health state can only be rolled up between objects managed by the same agent unless the source object is managed by the Root Management Server. Groups and classes used for health rollup are typically unhosted. This means that they are managed by the RMS so that they can roll up health from objects managed by different agents. A relationship can be discovered between objects managed by different agents, but any dependency monitor associated with that relationship will not work as expected.

Health rollup between agents



# Alerts from Monitors

Monitors can be configured to create an alert when they change from a healthy state to a warning state or a critical state. This alerting can be enabled on a monitor but just enabling the option, but other details of the alert should be considered.

## Alert Name

The name of the alert is a single line of static text and cannot include any variables.

## Alert Description

The alert description may have several lines of text that includes static text or variables. The most common kind of variable in the alert description will be $Data variables to include different information from the monitor’s data source in the description of the alert. The properties that are available will depend on the kind of data source being used. Each section of [Data Sources](#z6ab948a1cbfb4f65ad7fd4cb36d7f0fd) includes a list of the properties available for different data sources. The following table provides the syntax and examples of variables in monitor alerts created from different data sources:

|  |  |  |
| --- | --- | --- |
| Data Source | Syntax | Examples |
| Windows Event | $Data/Context/<Property Name>$ | $Data/Context/EventDescription$ |
| $Data/Context/Params/Param[#]$ | $Data/Context/Params/Param[2]$ |
| Text Log | $Data/Context/<Property Name>$ | $Data/Context/LogFileName$ |
| $Data/Context/Params/Param[1]$ | $Data/Context/Params/Param[1]$ |
| Delimited Text Log | $Data/Context/<Property Name>$ | $Data/Context/LogFileName$ |
| $Data/Context/Params/Param[#]$ | $Data/Context/Params/Param[2]$ |
| WMI Event | $Data/Context/Collection[@Name='<TargetInstance|PreviousInstance>']/Property[@Name='<PropertyName>']$ | $Data/Context/Collection[@Name=’TargetInstance’]/Property[@Name='Name']$ |
| Windows Performance | $Data/Context/<PropertyName>]$ | $Data/Context/Value$ |
| WMI Performance | $Data/Context/<PropertyName>]$ | $Data/Context/Value$ |
| Monitoring Script | $Data/Context/Property[@Name='<PropertyName>']$ | $Data/Context/Property[@Name='Result'>']$ |

## Priority and Severity

The Alert severity defines the alert as an Information, Warning, or Critical alert. This severity does not have to match the severity of the health state triggering the alert. The severity of the alert is identified by an icon in the Operations console and is used by views and notification subscriptions. The alert priority is inaccessible in the Operations console but is used primarily for notification subscriptions.

## Alert Suppression

Alert suppression is not required for monitors because alerts are only created when the monitor changes state. For example, consider a monitor that samples a performance counter on a regular interval. When the threshold is exceeded, the monitor changes to a critical state and creates an alert. The next time that the monitor samples the counter its value still exceeds the threshold. The monitor does not change state because that monitor is already in a critical state. Because the monitor does not change state, no alert is created.

## Automatic Alert Resolution

Monitors that create alerts can be configured to automatically resolve the alert when the monitor returns to a healthy state. This means that any unresolved alert for the monitor represents a problem that still exists. There is no configuration this requirement other than confirming the option that automatic resolution be performed.

## See Also

[Data Sources](#z6ab948a1cbfb4f65ad7fd4cb36d7f0fd)

[Unit Monitors](#z9130c17a58234046a7c719740c0d0ab9)

# Diagnostics and Recoveries

Diagnostics and recoveries are workflows that run when a monitor changes state. Diagnostics collect additional information about the detected problem. Recoveries try to resolve the problem. Each will typically run a command or script that outputs information displayed in the Health Explorer in the Operations Console.

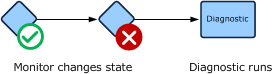
The Operations console provides a wizard for creating both diagnostics and recoveries. The Authoring console lets both diagnostics and recoveries be created from the properties of a monitor, although the individual modules must be selected and configured. Complete details on the concepts and use of modules in a management pack are provided in the [Composition](#z94c023257fa84110b5234f4b8b2f927c) section of this guide. This section will provide a simplified discussion of these concepts specific to diagnostics and recoveries.

## Diagnostics

Diagnostics are workflows that run after a monitor changes state and try to collect additional information about the issue. This information is provided to the user with the state change history in the properties of the monitor. If the enabled property of the diagnostic is set to true, then it is run automatically when the monitor changes state. If the disabled property of the diagnostic is set to false, then a link is provided to the user in the Operations console that they can click to run the diagnostic.

Diagnostics are not intended to make any changes to the application or system that they are running on. Because they are running a script or command, however, there is no way for Operations Manager to make sure that these changes are not being made. It is the responsibility of the management pack author to make sure that no such changes are being made. If changes to the application or system are required, then a recovery should be used.

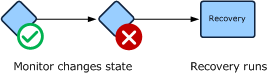
Diagnostic



## Recoveries

Recoveries are workflows that run after a monitor changes state. Recoveries try to correct the issue, and return the monitor to a healthy state. Any output from the recovery is provided to the user with the state change history in the properties of the monitor that the diagnostic is associated with. If the enabled property of the recovery is set to true, then the recovery is run automatically when the monitor changes state. If the disabled property of the recovery is set to false, then a link is provided to the user in the Operations console that they can click to run the recovery.

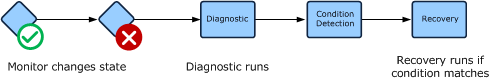
Recovery



Recoveries are intended to change the application or system that they are running on. Therefore, significant consideration should be taken before they are configured to run automatically. A poorly planned recovery script could actually cause more problems than it is intended to help. For example, the recovery may temporarily correct issue but not the underlying cause of the problem. In this case, the monitor may return to a healthy state because of the recovery, but then return to an error state when the issue is again detected. The recovery would then run in response, and so on, until a user is able to intervene and correct the root problem. This kind of problem can be limited by disabling the recovery so that the user must run it instead of it running automatically.

Another option is to run a recovery after a diagnostic. Using this strategy the diagnostic first collects additional information about the issue which the recovery then uses that information to determine whether it should run. This is implemented in the Authoring console by configuring condition detection with the recovery. The condition detection uses output from the diagnostic to determine whether it should run.

Recovery running after a diagnostic



### Recalculating State

If a recovery is successful, then the monitor should return to a healthy state the next time that the monitor detects the required information. If the monitor runs a scheduled script for example, then the monitor will return to healthy the next time that the script runs and the monitors detects the criteria for a healthy state. If the monitor relies on an event for its healthy state, the application is expected to create the required event in response to the recovery successfully correcting the problem. If a monitor is configured to use a manual reset though, then the user will still be required to manually set it to a healthy state.

A recovery can be configured to recalculate the state of the monitor immediately after it runs. This option has the same effect as the user selecting Recalculate Health for the monitor in the Operations console. Recalculating state only has an effect on monitors that run on a schedule such as a script and that have on demand detection defined. If the monitor does not have on demand detection defined, then the option has no effect. The advantage of configuring a monitor to recalculate state is that it can return the monitor to a healthy state immediately instead of waiting for the schedule.

## When Diagnostics and Recoveries Run

Diagnostics and recoveries run in response to a change in a monitor’s state. They do not run every time their specified state is detected; only when the monitor changes to that state from another state. For example, a particular monitor might run a script every few minutes to test a particular feature of an application. A diagnostic could be defined to run when the monitor’s health is critical. The diagnostic will only run when the monitor changes from a healthy or a warning state to a critical state when the script first identifies a problem. Assuming the problem has not been corrected the next time that the script runs, the monitor will still be in a critical state. The diagnostic will not run though because no state change will have occurred. It will only run when the monitor returns to a healthy state and then changes again to a critical state.

## Modules

The concept of modules in a workflow is discussed in detail in the [Composition](#z94c023257fa84110b5234f4b8b2f927c) section. Frequently, the Authoring console can provide wizards and other means of assistance to help the author with this complexity. The wizards create workflows by using the same modules, although the user is provided with an easier interface. Recoveries and diagnostics can be configured from the Authoring console in the properties dialog box of the monitor, but their modules must be configured manually.

The following table provides a listing of common modules that are used in diagnostics and recoveries:

|  |  |  |
| --- | --- | --- |
| Module | Used In | Function |
| Microsoft.Windows.ScriptProbeAction | Action | Runs a script in either VBScript or Jscript. |
| Microsoft.Windows.PowerShellPropertyBagProbe | Action | Runs a script in Windows PowerShell. |
| System.CommandExecuter | Action | Runs an executable on a command line. |
| System.ExpressionFilter | Condition Detection | Evaluates output from a diagnostic to determine whether recovery should be run. |

Examples of the use of these modules are provided in the [Building a Health Model](#z1de16bb513ac4e90be911543c19f4f70) section of this guide.

# Rules

Rules in System Center Operations Manager 2007 use the same [Data Sources](#z6ab948a1cbfb4f65ad7fd4cb36d7f0fd) as monitors but provide different functionality. While a monitor changes state, a rule performs one of the following three functions:

 Create an alert that is not related to health state.

 Collect performance or event data for analysis and reporting.

 Run a script or command.

## In This Section

[Alert Rules](#z0ee45649328c4c039a427258b9e3edce)

|  |
| --- |
| Rules that create alerts not related to the health state of an managed object. |

[Collection Rules](#za3da9da54d794c808476b3e8ad20d057)

|  |
| --- |
| Rules that collect event or performance data. |

# Alert Rules

Alert rules create alerts in response to particular conditions detected in a data source. This is the same kind of alert that is created by a monitor when it changes state. Monitors and alert rules use the same data sources and typically the same kind of logic for determining whether an error has occurred.

A monitor that creates an alert is generally preferred over an alert rule identifying the same issue for the following reasons:

 Monitors set a health state on the target object in addition to creating the alert. This identifies the category of the detected problem, the application component affected, and its effect on the overall health of the application. The health state is also recorded for availability reports that provide historical record of the availability of the application.

 Alerts created by monitors can be automatically resolved when the application returns to a healthy state. Alerts created by rules cannot be automatically resolved because there is no method of determining the healthy state.

The situations when a rule should be used instead of a monitor to create an alert are as follows:

 The problem being detected does not relate to the health of the application. For example, an application may perform an automated nightly backup. If the backup fails, then an alert should be created to inform users of this condition. The application though is still completely healthy and should not record a negative health state.

 A monitor is cannot determine when the detected problem was resolved. One of the options to this condition is to use a rule to create an alert instead of a monitor. This situation is discussed with additional options in the [Event Monitors](#z9a556512ab434d75aa4dd6958c0f3457) section of this guide.

## Event Alert Rules

Alert rules can be created for each event data source. The criteria that is specified to determine when an alert should be created is the same as the criteria for a state change in the event monitors.

## Performance Alert Rules

The Authoring Console provides no wizards for creating an alert rule based on a performance counter. A monitor should be used instead because a success condition is usually detectable from a performance counter and is usually related to some health state of the target class. Alert rules based on a performance counter can be created, although they must be done with a custom rule.

## Scripting Alert Rules

The Authoring Console provides no wizards for creating an alert rule based on a script. A monitor should be used instead because a script will typically provide a return value for both and error and a healthy state in such a way that a success condition is usually detectable and related to some health state of the target class. Alert rules based on a script can be created, although they must be done with a custom rule.

## Alerts from Rules

### Alert Name

The name of the alert is a single line of static text and cannot include any variables.

### Alert Description

The alert description may have several lines of text that includes static text or variables. The most common kind of variable in the alert description will be $Data variables to include different information from the rule’s data source in the description of the alert. The properties that are available will depend on the kind of data source being used. Each section of [Data Sources](#z6ab948a1cbfb4f65ad7fd4cb36d7f0fd) includes a list of the properties available for different data sources. The following table provides syntax and examples of variables in rule alerts created from different data sources:

|  |  |  |
| --- | --- | --- |
| Data Source | Syntax | Variable |
| Windows Event | $Data/<Property Name>$ | $Data/EventDescription$ |
| $Data/Params/Param[#]$ | $Data/Params/Param[2]$ |
| Text Log | $Data/EventData/DataItem/<PropertyName>$ | $Data/EventData/DataItem/LogFileName$ |
| $Data/EventData/DataItem/Params/Param[1]$ | $Data/EventData/DataItem/Params/Param[1]$ |
| Delimited Text Log | $Data/EventData/DataItem/<PropertyName>$ | $Data/EventData/DataItem/LogFileName$ |
| $Data/EventData/DataItem/Params/Param[#]$ | $Data/EventData/DataItem/Params/Param[2]$ |
| WMI Event | $Data/EventData/DataItem/Collection[@Name='<TargetInstance | PreviousInstance>']/Property[@Name='<PropertyName>']$ | $Data/EventData/DataItem/Collection[@Name='TargetInstance']/Property[@Name='Name']$ |
| Syslog Event | $Data/EventData/DataItem/<PropertyName>$ | $Data/EventData/DataItem/Facility$ |

### Priority and Severity

The Alert severity defines the alert as either Information, Warning, or Critical. This severity does not have to match the severity of the health state triggering the alert. The severity of the alert is identified by an icon in the Operations console and is used by views and notification subscriptions. The alert priority is inaccessible in the Operations console but is used primarily for notification subscriptions.

### Alert Suppression

Alert suppression refers to logic that is defined on alert rules to suppress the creating an alert when a corresponding alert is still open. This prevents alert storms where multiple alerts are created for the same issue. Because the issue has already been identified with an open alert, creation of additional alert creates unnecessary noise with minimal value. When the condition for an alerting rule is met but an existing alert is already open, instead of creating an additional alert suppression will increase the repeat count of the existing alert.

In order to define suppression on an alerting rule, the fields must be specified that identify a matching alert. Before an alerting rule creates a new alert, it will check whether an open alert exists with values for the fields that are defined for suppression that match the fields of the new alert. If an alert with matching values for each of these fields is open, then a new alert is not created.

The minimum number of fields that uniquely identify the alert should be specified for alert suppression. This is typically the computer name in addition to the fields used for the criteria of the rule. For example, suppression on event rules can frequently be achieved by using the following fields:

 Logging Computer

 Event Source

 Event Number

If the rule is targeted at a class that has multiple instances on the agent, however, then a parameter might be required to uniquely identify the event in the criteria of the rule. If this is the case, then the same parameter should be specified in the alert suppression.

### Automatic Alert Resolution

Automatic resolution cannot be performed on alerts that are created from a rule since a rule has no mechanism for determining that

# Collection Rules

Collection Rules are used to collect data into the Operations Manager database and data warehouse for analysis in views and into the Data Warehouse for use in reports. Even though a particular data source might be used for a monitor or alerting rule, a collection rule is still required if its information is to be available for this analysis. For example, a monitor may sample a performance counter on a regular interval and compare the value to a threshold in order to set its state. It will not store this information however, so a collection rule is also required using the same performance counter if the data is required to be views.

## Collection rules described

Collection rules store data as either events or performance data, and any data source must have its data mapped into one of these data types. Each kind of collection rule is described in the following sections:

# Event Collection Rules

Events in System Center Operations Manager 2007 can be collected from any of the data sources detailed in the [Events](#z7ddb06a4360e487fa48d5a2a615294f6) section of this guide. For most data sources, all that is required is the criteria for which events are to be collected. This criteria is specified using the properties available to the particular data source being used.

## Script Based Event Collection

Script based event collection runs a monitoring script on a regular schedule and stores the results as an event. The monitoring script returns a property bag as described in the [Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8) section of this guide. The performance collection rule maps values of the property bag into properties of the event by using $Data variables referring to different values in the property bag.

Event properties that the rule may populate are listed in the following table:

|  |  |
| --- | --- |
| Property | Description |
| Computer | The computer logging the event. Typically uses the PrincipalName property of the target object’s host computer. |
| Event source | The source of the event. Typically either a static string or a value from the property bag. |
| Event log | The name of the event log. Typically either a static string or a value from the property bag. |
| Event ID | The number of the event. Typically either a static string or a value from the property bag. |
| Category | The category of the event. |
| Level | The level of the event. Typically selected from the list of available options. |
| Parameters | Multiple values that contains data that does fit into the other properties. Typically one or more values from the property bag. |

## See Also

[Events](#z7ddb06a4360e487fa48d5a2a615294f6)

[Alert Rules](#z0ee45649328c4c039a427258b9e3edce)

[How to create a script-based event collection rule](#z7a3d883db9404b608b162199160b0d15)

# Performance Collection Rules

Performance data can be collected in System Center Operations Manager 2007 from any of the data sources detailed in the [Performance Data](#z336bae3fc9794e7ca162c626c55e6d94) section of this guide. This section provides the details required for specifying the source of the data and the properties available for specifying the criteria for the data to be collected.

## Script Based Performance Collection

Script based performance collection runs a monitoring script on a regular schedule and stores the results as performance data. The monitoring script returns a property bag as described in the [Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8) section of this guide. The performance collection rule maps values of the property bag into properties of the performance data by using $Data variables referring to different values in the property bag.

Performance data properties that the rule must populate are listed in the following table:

|  |  |
| --- | --- |
| Property | Description |
| Object | Name of the performance object. This is typically a static value but may be a $Data variable to retrieve a value from the property bag returned from the script. |
| Counter | Name of the performance counter. This is typically a static value but may be a $Data variable to retrieve a value from the property bag returned from the script. |
| Instance | Name of the instance if it is specified. If the target of the rule has a single instance on the agent, the instance name may not be specified. If the target of the rule has multiple instances on the agent, an instance name should be specified by using a $Target variable to retrieve the value of a unique property to identify the target object. |
| Value | The numeric value to store. This is typically a $Data variable to retrieve a value from the property bag returned from the script. |

## Optimized Collection

Performance collection rules based on Windows performance counters can be configured to perform optimized collection. Optimized collection reduces the space that is required by only sampling those performance counters that differ significantly from a previously sampled counter.

When optimized collection is specified, a tolerance must be specified that indicates the value that the sampled data must differ from the previously sampled value for the data to be stored. This tolerance can be either an absolute number or a percentage. An absolute tolerance evaluates the difference between the current and the last counter. A percentage tolerance evaluates the difference as a percentage of the previously sampled value.

An example of optimized collection is the Microsoft.Windows.Server.2008.LogicalDisk.FreeMB.Collection rule in the Microsoft Windows Server 2008 Monitoring management pack. This rule collects the free space in MB on a logical disk every 5 minutes. Free disk space is a value that typically changes gradually, and under most conditions collecting it with this frequency would create an excess number of sampled counters with minimal value. Increasing the frequency of collection though would introduce the chance of missing those periods where a sudden change in the value did occur.

This rule uses optimized collection specifying an absolute value of 100 MB. This means that the counter is sampled every 5 minutes, but the value is only stored if it differs from the last stored value by 100 MB. This still lets it to perform its sampling at a fairly frequent rate but significantly reduces its storage requirements by reducing the number of unnecessary data points.

## See Also

[Performance Data](#z336bae3fc9794e7ca162c626c55e6d94)

# Tasks

Tasks in System Center Operations Manager 2007 are actions that can be run on demand by the user. Depending on the kind of task, the action may run either on the user’s local workstation or on one or more specified agents.

## In This Section

[Console Tasks](#zd5d1e2c81e56456e994f20cb2e364e52)

|  |
| --- |
| Tasks that run on the user’s workstation using the current user’s credentials. |

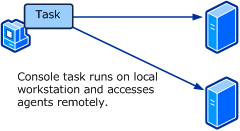
[Agent Tasks](#za0a4f9ac9808425a81e43068f7c813db)

|  |
| --- |
| Tasks that run on the agent computer using the credentials of the specified user profile. |

# Console Tasks

Console tasks run on the workstation where the Operations Console is running and uses the same credentials as the logged on user. The application that is run by the task must be installed on the workstation.

Console task



Console tasks are useful for running administration consoles or other tools that remotely access application components. These will usually be executable programs that run outside the Operations console. $Target variables can be used to provide values of properties on the target object or one of its parents to be included on the command line.

For example, an administration console might require the name of the server that is running the application. The target for the task could be a class based on Windows Computer Role representing the application installation on the server. The task would only be available when an instance of the target class is selected. Because the class is hosted by Windows Computer, the PrincipalName property could be provided on the command line.

# Agent Tasks

Agent tasks in System Center Operations Manager 2007 are run on the agent computer where the target object is managed. An agent task can be a script or an executable program run from a command line. If an executable program is used, the application must be installed on the agent computer.

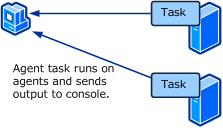
Agent tasks are useful for performing actions on the agent computer or for retrieving information for the user. They provide the following capabilities:

 Run a script or command locally on the agent computer without logging on to the computer interactively.

 Run a script or command on multiple agents with a single action.

 Run a script or command by using local user credentials with permissions not available to the user.

Agent Task



## Credentials

Unless otherwise specified, tasks run under the credentials of the Default Action Account on the agent computer. This account typically has sufficient privileges for accessing most application components, even if the user running the task does not have these user rights. If the task is required to perform an action requiring other credentials, such as accessing an external data source, then a secure reference can be created in the management pack for running the task. Or, the user can be required to specify credentials when they run the task.

## Output

Any output sent to the standard out stream (StdOut) from the script or command is provided to the user as Task Output in the Operations Console. Command line programs will typically output information to this stream. Scripts should output information by using commands such as WScript.Echo to provide this information.

## Kinds of agent tasks

Whether an agent task is configured to run a command or a script, it must specify whether it uses a probe action or a write action. This specifies the kind of module that is used by the task as described in the [Composition](#z94c023257fa84110b5234f4b8b2f927c) section of this guide. There is no difference in the configuration or the capabilities between two. Probe action modules are intended to gather information without making any change to the agent computer, whereas write action modules are intended to make some change or perform an action on the agent.

# Health Model Design Overview

Designing a health model System Center Operations Manager 2007 for an application consists of determining the required set of monitors to accurately measure the health of each class in the service model. Dependency monitors are defined to order to roll up health between classes and provide an overall application health. Rules are added to collect valuable information for analysis and reporting.

The following sections provide a basic process to help you design a health model to monitor an application. This process is not intended to apply to all applications, although this assumes a service model that uses the basic process presented in the [Service Model Design Overview](#z879b344ff01b42d59d45634f474a2a56) section of this guide. The basic principles followed in both processes should apply to most applications monitored by a management pack.

## In This Section

[Failure Mode Analysis](#zdce0ae25dd7c4bd98a34bf5439b8630c)

|  |
| --- |
| Overview of a process for software developers to follow what they include monitoring instrumentation in an application. |

[Defining Monitors](#z2d2d2ff835744f5a9d9f06679497b060)

|  |
| --- |
| Basic process for defining a set of monitors to accurately measure the health of each class in the service model. |

[Defining Rules](#zba2b17183b33489f918364b23057db95)

|  |
| --- |
| Basic process for defining rules to augment health monitoring by collecting data for analysis and reporting. |

[Practices to Avoid](#z8c93079050a7423c994202de05b3f42a)

|  |
| --- |
| Common practices to avoid when you design a health model. |

# Failure Mode Analysis

Failure Mode Analysis is a process for identifying different issues that an application may experience and providing instrumentation within the application that may be used for monitoring. It is primarily used by software developers in designing their application, but the basic concepts may also be valuable to the management pack author designing a management pack. The process is discussed in detail in [MP University](http://go.microsoft.com/fwlink/?LinkID=164900&clcid=0x409) which is a training program for designing and building management packs targeted at software developers. It is briefly described here, and the training program should be referenced for more information.

This topic presumes that the management pack author has no control over changes to the application itself, and the health model must be designed according to the information that the application makes available. The quality of a management pack though is directly affected by the quality this available information. If an application has features that cannot be monitored, then the management pack will be unable to provide a complete health measurement. Even if a basic failure can be detected, the management pack may be unavailable to identify the underlying cause of this failure unless the application makes that information available. If the management pack author can collaborate with software developers, then they may be able to influence modifications to the application itself that allow it to be monitored more effectively.

## Basic Concept

The basic concept of failure mode analysis is to analyze what can fail in an application and provide a means for detecting that failure. By implementing such instrumentation within the application, a management pack can accurately measure health of the application and detect any potential failures. Failure mode analysis resembles threat modeling because it is a proactive effort to identify potential weaknesses in the application. It is an attempt to identify the components of the application that are vulnerable to failure, what those failures may be, and how they might be detected.

## Steps in Failure Mode Analysis

### List what can go wrong

This list should go beyond software issues and also consider specific implementations and issues that can arise in the environment. It includes each component of the application and each component that the application relies on – such as network connectivity, server resources, and dependent software components. The underlying cause should be considered in this analysis to distinguish a failure from a symptom of that failure. For example, a database may run out of space causing other errors in the application.

As soon as a complete list is established, it should be prioritized according to the following criteria. This will assist in prioritizing instrumentation for the different scenarios.

 Probably of the failure occurring

 Impact to overall system health

 Potential cost to the customer for a failure

### Identify a detection strategy for each failure mode

Identifying a potential failure is useless if there is no way of detecting it. Each failure mode needs at least one means of detection, and especially high impact issues should have multiple means of detection. This detection could be a predictable error in code generating an event or may need additional code to continuously watch for a particular operation.

### Add detection elements to application code

Each of the detections has to be added to the code of the application in order to expose this information to a management pack. Some of these elements may be monitors watching for particular event to occur, or they may be probes periodically performing some test or collecting some information. The result of the detection should be information that can be accessed by a management pack. This might be an event in the Windows event log or a performance counter providing a numeric measurement of the application’s health.

### Plan management pack content

The end result of failure mode analysis is to design the management pack based on the information exposed by the application. This will include defining monitors that use the events and performance counters exposed by the application in order to set appropriate health states on classes defined in the application’s service model.

## Operational failures

The following table lists a set of operational failures that impact most applications. This list can be referenced in identifying a set of potential failures for a particular application.

|  |  |
| --- | --- |
| Category | Failure Mode |
| Access Control Lists | Insufficient permissions accessing resource |
| Run as account has expired |
| Run as account locked out |
| Run as account soon to expire |
| ACL on configuration resource too permissive |
| Capacity | Disk is full |
| Disk utilization above threshold |
| Critical resource starved |
| Critical resource above threshold |
| Capacity trend predicts coming failure |
| Queue reads failing behind queue writes |
| Restart under load fails |
| Component is hung |
| Configuration | Configuration file missing |
| Configuration file corrupt |
| Connection string incorrect |
| Critical setting missing or incorrect |
| Database | Database offline |
| Login denied |
| Execute permission denied |
| Database doesn’t exist |
| Timeout on database connection |
| Index corrupt |
| Table or database corrupt |
| Transaction log full |
| SPID blockage |
| Incorrect results in configuration table |
| Network | Access to critical network resource denied |
| Critical network resource timing out |
| Average network latency exceeds tolerance |
| Throughput diminished causing backlog |
| Packet loss rate on subnet exceeds tolerance |
| Resend rate on subnet exceeds tolerance |
| Latency exceeds tolerance |
| Transaction | Transaction rate below minimum threshold |
| Transaction response time above threshold |
| Transaction rate exceeds engineering limits |
| Incorrect response being returned |
| Failure response rate exceeds threshold |

With these base causes for the failure identified, the part of the application that performs this database connection should include code to detect each failure and provide a unique event that may be accessed by monitors in the management pack. If the application just reports that the database connection failed, then the management pack cannot identify the underlying cause. Only if the application provides this detailed information can the management pack accurately detect the issue and identify the cause to the user.

# Defining Monitors

The first step in designing the health model for an application is defining monitors for each class in the service model. The goal is to provide a combination of different kinds of monitors to appropriately measure the health of each managed object. If the set of monitors are designed correctly, then any issue with the application will result in one or more monitors changing from a healthy state to warning or critical. If there are flaws in the design of the monitors, then the actual health of the application may not be accurately detected. An issue may occur in the application that is not recognized by a monitor, or a monitor may change to a negative state when the application is not actually experiencing a problem.

## Define Unit Monitors

A combination of different kinds of monitors will frequently be required to accurately measure the health state of a particular class. The questions that can be asked for each class to define its monitors are as follows:

 What functions does the object perform and how can each function be tested? Does the application provide this information or is a synthetic transaction required to test the functionality?

 What information does the application create that might indicate a loss in functionality? Where is the information stored? What component of the application does it apply to? What, if any, indicators are there that the problem is corrected?

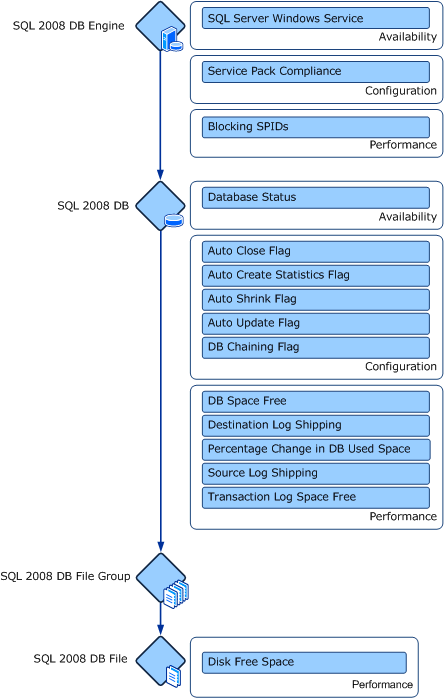
 How can the performance of the application be measured? Does the application make this information available or is a synthetic transaction required to collect the information?

 Are there any configuration requirements that could affect the operation of the application? Can this information be collected?

The answers to these questions will help in the definition of unit monitors to test different aspects of the application. It is important to define monitoring requirements for an application even if there is no identified ways to implement one or more of the required monitors. If a feature of an application provides no means of monitoring, then the application may experience an issue that the management pack will be unable to detect. This issue should at least be documented so that owners of the application understand the limitations of the monitoring being implemented. This information can also be used to work with developers to potentially change the application to make such monitoring possible.

As an example, the health model for the SQL Server 2008 DB class is shown here. This includes the SQL Server 2008 DB class itself in addition to the monitors for the SQL Server 2008 DB Engine class that hosts it and the SQL Server 2008 DB File class that it hosts. A diagram such as this can be useful in picturing the health of a particular class in addition to the classes directly related to it.

SQL Server 2008 DB health model



The database class itself is monitored by a combination of monitors by using events, scripts, and performance data to measure the health of each database. Availability status of the database cannot be retrieved through a simple method so that a script is required that connects to the database engine and validates the connection to each database. The script determines a status for the database identifying whether it is available for connections. If a connection to the database cannot be made for any reason, then this monitor will change to a warning or critical state.

Configuration monitors are defined for both the SQL Server 2008 DB class and the SQL Server 2008 DB Engine class. For the database engine, the SQL Server service pack level is retrieved through WMI to determine whether it is at a required level. The user is expected to use an override to define the required level for their environment. Configuration monitors for each database are defined by using a script that retrieves different properties of the database and compares each to recommended values. Configuration of the database or database engine does not necessarily affect availability or performance, but providing monitors for these properties enables the user to identify configurations that do not match required standards for their environment and that could lead to eventual lead to issues with availability or performance.

### Diagnostics and Recoveries

As soon as monitors are defined, diagnostics and recoveries for each can be considered. Most monitors will not have either. Diagnostics are only defined if there is additional information to collect about the detected issue. Many monitors already contain all available information, or additional information must be collected manually by the user. Diagnostics should only be defined when additional information about the issue that may be useful to the user can be retrieved programmatically.

As discussed in the [Diagnostics and Recoveries](#z6d472fd061a64c7ca44f19ddb8e3cd6e) section of this guide, recoveries should only be defined after careful consideration. Because recoveries try to make some change to the external environment, they can cause issues in addition to the one they are responding to. The strategies discussed in that section of configuring recoveries to run manually or after a diagnostic can be used to lessen this risk. An automated recovery should only be configured in response to a monitor if the following two questions can be answered affirmatively.

 Has the specific issue that the recovery is intended to correct been correctly identified?

 Does the recovery have a high degree of probability that it will correct the issue it is designed to correct without causing adverse side effects.

These questions may seem common sense, but realize that there may be multiple root causes for the issue detected a monitor, and each may require a different kind of recovery. Before a recovery is configured to run automatically, it should be verified that it will only run in response to the cause that it is designed for.

One strategy for designing diagnostics and recoveries is to delay their definition until the management pack has run in production for a while. This allows monitors to change state in response to actual issues with the application. Analysis of these incidents and the manual steps that were taken to correct them can lead to automated responses defined in diagnostics and recoveries.

## Define Aggregate Monitors

Most applications will be adequately managed by using the standard set of aggregate monitors. Additional aggregate monitors may be added to a management pack to group related unit monitors targeted at a common class. This is performed either to make a large number of monitors more manageable or to provide a consolidated health state in a particular category.

For example, the Microsoft Windows DNS 2008 Server management pack includes an aggregate monitor called Resolution Rollup that is used for unit monitors that perform test resolutions and another called Services State Rollup used for unit monitors related to the state of DNS services. These aggregate monitors provide a combined health state for each set of related monitors and the Health Explorer for the state of the DNS Server 2008 class.

## Define Dependency Monitors

As soon as unit monitors are defined to measure the health of individual classes and each are configured underneath a standard or custom aggregate monitor, then dependency monitors can be defined to roll up health between instances of different classes.

### Hosted Classes

Any classes that use Windows Computer Role as a base class will automatically have their health roll up to Windows Computer. This is because the Microsoft Windows Library management pack includes a dependency monitor for Windows Computer Role targeted at Windows Computer. No dependency monitor has to be defined in this case.

Classes that use Windows Local Application as a base class will not automatically have their health roll up to Windows Computer. If the health of these classes should rollup to the hosting computer, then a dependency monitor has to be created.

Classes that use Windows Application Component as a base class and are hosted by another class in the application’s service model require a dependency monitor if their health should be rolled up to the hosting class. The hosting relationship that is required for the dependency monitor will already be in place.

The rollup policy for these dependency monitors will typically be worst state so that any instance of the Application Component class in an error state will cause a corresponding error state on the parent object. Best state and percentage though may be used for those applications requiring different logic.

### Health Rollups

A dependency monitor will typically be defined for each containment relationship between any health rollup classes, if such classes are part of the application’s service model. The purpose for these classes is typically to roll up health for a set of objects, and this function is not performed without a dependency monitor. These dependency monitors will use different rollup policies based on their requirements. Worst state is the most common policy that is used, although best state and percent policies are common for health rollups.

# Defining Rules

After monitors and health state are defined for each class in the service model of a System Center Operations Manager 2007 management pack, rules can be defined.

## Alerting Rules

Alerting rules are used either when a monitor cannot reasonably be defined for a particular issue or when the issue does not relate to the health state of a particular class. The most common reason that a monitor cannot be defined is that there is no reasonable means of identifying that the issue was corrected. This is required to define the healthy state of the monitor. One strategy to address this challenge is using an alerting rule instead of a monitor.

Instead of creating an alerting rule, a monitor that uses a timer reset can be considered. We do not recommend a manual reset, especially targeted at a class that has multiple instances, because of the requirement that a user must always intervene to return the monitor to a healthy state. A timer reset monitor still influences the health state of the target object and will automatically reset to a healthy state if a user does not manually reset it in a specified time.

The other use of alert rules is to identify issues that do not relate to a health state. For example, the application might perform a data replication regularly. An alert should be created if the transfer fails, although the application is still fully functional. This issue is best identified through an alerting rule instead of a monitor.

A strategy that can be used to identify alert rules is to analyze all events created by the application that were not used in a monitor. These may be Windows events or may be provided in any of the event data sources previously discussed. Such events will typically identify errors in the application. Any events not used by a monitor are candidates for an alerting rule.

### Minimizing Noise

A strategy of just creating an alerting rule for each event created by the application without analysis is not a best practice because excessive alerts can represent a source of noise in the Operations Manager environment with minimal value. Before you create any alerting rule, consideration should be given to whether the issue that is identified by the event is significant enough to create an alert. If the issue is minor, the event may just be collected for later analysis.

Analysis should also be performed to determine whether an event represents a unique issue not identified by another rule or monitor. A single application issue may have several symptoms generating multiple kinds of error events. There is minimal value in creating multiple alerts from the same root problem. Such noise can create too much work for the user and obscure alerts that identify other issues.

## Collection Rules

Collection rules augment health monitoring by collecting information for reporting and analysis in the Operations console. The most common collection rules will be those collecting performance data. This may be any Windows performance counters that are provided by the application or could be response times and other numeric data that is created by monitoring scripts.

The first data that should be considered for collection is any performance data that is used in monitors. The monitor sets health state based on comparing the numeric data to a threshold, but it does not store the data for analysis. Having trending and historical data to support the monitor will typically be useful for the end-user. Additional performance data that might be available should be defined taking into consideration storage requirements. If performance data is of minimal value in daily monitoring then it may not be worth the space that is required for its storage.

## See Also

[Alert Rules](#z0ee45649328c4c039a427258b9e3edce)

[Collection Rules](#za3da9da54d794c808476b3e8ad20d057)

# Practices to Avoid

## Focus on the easy–to-monitor instead of the required

We recognize the natural tendency to start a health model design for an application by analyzing the monitoring information made available by the application through events and other easily obtained information. However, this approach assumes that the application is proactively identifying all issues and may leave a significant potential for an application issue to go undetected. If an application has a problem that does not generate an event or other information, then the issue will require a synthetic transaction or other way of identification.

One consistent strategy is to analyze the different features and potential issues for each classe defined in the service model. A process such as [Failure Mode Analysis](#zdce0ae25dd7c4bd98a34bf5439b8630c) can help in this analysis. As soon as the monitoring requirements are identified, appropriate monitors and rules can be defined to address those requirements. This strategy encourages development of synthetic transactions and other proactive monitoring techniques in order to obtain information that the application does not make available on its own.

## Running monitors and rules too frequently

Monitors and rules that use a script or WMI query that run too frequently can result in excess overhead on the agent. The overhead will vary, depending on the script or query and the number of other scripts and queries in the management pack. A general guideline is not to run a script or query at any interval more frequently than 5 minutes. Even with this guidance, the script or query should not be run any more frequently than is required for needed monitoring.

For collection rules, too frequent collection can result in significantly increased storage requirements and return minimal value. Although a single performance value is fairly small, storage requirements quickly accumulate for collection targeted at multiple instances across multiple agents. We recommend that you never have a collection frequency less than 5 minutes for counters with significant volatility. A frequency of 15 minutes is effective for counters with less variance. It also a best practice to use optimized collection for any counter that is not expected to consistently have a significantly different value between consecutive samples.

## Multiple scripts running at the same time

If several monitors and rules that are running scripts are defined to run on the same interval, then there is a potential for them to consistently run at the same time. Every time that the interval arrives, all the scripts run, and a noticeable affect can be observed on the agent processor. One method to avoid this condition is to use a different synchronization time for each monitor or rule to make sure that they run at different times.

## Targeting classes with multiple instances

As is discussed in the [Targets](#z27daded3583549aaac6fea8cba48a2dd) section of this guide, a workflow targeted at a class that has multiple instances on an agent will have multiple copies of the workflow running on the same agent. Such workflows will typically require additional criteria by using a $Target variable to uniquely identify the particular instance. Failure to provide unique criteria can result in multiple alerts being created for a single event, or all instances of a particular class entering an error state instead of a single instance.

## See Also

[Targets](#z27daded3583549aaac6fea8cba48a2dd)

[Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8)

[Health Model Design Overview](#z1360d77d18064e85aac24aee6a1bc8f3)

# Building a Health Model

The following sections provide walkthroughs of the activities required to implement a health model in System Center Operations Manager 2007.

## In This Section

[Creating Monitors](#zb85eec915ae34891a7dadc0f08fe570f)

|  |
| --- |
| Create different types of monitors by using the Operations Manager Authoring console. |

[Creating Rules](#z12b5015aee6e44a9858e667054b988b3)

|  |
| --- |
| Create different types of rules by using the Operations Manager Authoring console. |

# Creating Monitors

The following procedures provide guidance on how to create monitors in the Operations Manager 2007 Authoring console.

## In This Section

[How to create an event monitor](#za74f6dde42c8497aab3ece252496f4aa)

|  |
| --- |
| Create a monitor from a single Windows event. |

[How to create an event monitor targeted at class with multiple instances](#z206b8da1e913476081f7e7edf5716806)

|  |
| --- |
| Create a monitor from a single Windows event monitoring a class with multiple instances. |

[How to create a repeating event monitor](#z904491de31024a608ad094c596261316)

|  |
| --- |
| Create a monitor from a single Windows event occurring multiple times within a specified timeframe. |

[How to create a monitor based on a script](#zfac2ebbdd83c4c18b2e7c567a2269d2b)

|  |
| --- |
| Create a monitor from data retrieved from a script. |

[How to create a WMI event monitor](#z2f1c2e873d5e4675b92f3ce6b23cfd83)

|  |
| --- |
| Create a monitor from a WMI event. |

[How to create a consecutive sample performance monitor](#z6c45887a846b47ad8dd544bd9db9d545)

|  |
| --- |
| Create a monitor from a consecutive samples of a performance counter. |

[How to create a dependency monitor](#z24abb0f34b9640d2892e13555c566386)

|  |
| --- |
| Create a monitor from a consecutive samples of a performance counter. |

# How to create an event monitor

The following procedure shows how to create an event monitor in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 Sets the monitor to a critical state when an event in the Application event log with an event source of MyApplication and an event number of 101 is detected.

 Sets the monitor to a healthy state when an event in the Application event log with an event source of MyApplication and an event number of 102 is detected.

To create an event monitor

|  |
| --- |
| 1. In the Authoring Console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand Computer Role 1, and then expand System.Health.EntityState.  3. Right-click System.Health.AvailabilityState, select New, select Windows Events, select Simple, and then select Event Reset.  4. On the General page, do the following:  a. In the ElementID box, type MyMP.Monitor.MyApplicationEventError.  b. In the Display Name box, type MyApplication Event Error.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Parent Monitor box, select System.Health.AvailabilityState.  e. In the Category box, select AvailabilityHealth. Click Next.  5. On the Event Log (Unhealthy Event) page, in the Log Name box, keep the default value of Application. Click Next.  6. On the Event Expression (Unhealthy Event) page, do the following:  a. For the Event ID value, type 101  b. For the Event Source value, type MyApplication  c. Click Next.  7. On the Event Log (Healthy Event) page, in the Log Name box, keep the default value of Application. Click Next.  8. On the Event Expression (Healthy Event) page, do the following:  a. For the Event ID value, type 102  b. For the Event Source value, type MyApplication  c. Click Finish.  9. Expand System.Health.AvailabilityState.  10. Right-click MyMP.Monitor.MyApplicationEventError and select Properties.  11. On the Health tab, for FirstEventRaised, change the Health State to Critical.  12. On the Alerting tab, check Generate alerts for this monitor.  13. Click OK. |

## See Also

[Events](#z7ddb06a4360e487fa48d5a2a615294f6)

[Event Monitors](#z9a556512ab434d75aa4dd6958c0f3457)

# How to create an event monitor targeted at class with multiple instances

The following procedure shows how to create an event monitor in the Operations Manager 2007 Authoring console that is targeted at a class that has multiple instances on the agent. Because multiple instances are expected, additional criteria is required that includes a property of the target object. This ensures that each copy of the monitor applies to only a single instance of the target class. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has one or more instances of MyApplicationComponent.

 Sets the monitor to a critical state when an event in the Application event log with an event source of MyApplication and an event number of 201 is detected.

 Sets the monitor to a healthy state when an event in the Application event log with an event source of MyApplication and an event number of 202 is detected.

 The name of the component is included in parameter 1 of the event.

To create an event monitor

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand My Application Component and then System.Health.EntityState.  3. Right-click System.Health.AvailabilityState, select New, select Windows Events, select Simple, and then select Event Reset.  4. On the General page, do the following:  a. In the ElementID box, type MyMP.Monitor.MyApplicationComponentEventError.  b. In the Display Name box, type My Application Component Event Error.  c. In the Target box, select MyMP.MyApplicationComponent.  d. In the Parent Monitor box, select System.Health.AvailabilityState.  e. In the Category box, select AvailabilityHealth. Click Next.  5. On the Event Log (Unhealthy Event) page, in the Log Name box, keep the default value of Application. Click Next.  6. On the Event Expression (Unhealthy Event) page, do the following:  a. For the Event ID value, type 201  b. For the Event Source value, type MyApplication  c. Click Insert.  d. Click the button to the right side of the Parameter Name box on the new entry.  e. Select Specify event specific parameter to use.  f. Specify 1 for the event parameter.  g. Click OK.  h. In the Operator box, select equals.  i. Click the button to the right side of the Value box and select ComponentName.  j. Click Next.  7. On the Event Log (Healthy Event) page, in the Log Name box, keep the default value of Application. Click Next.  8. On the Event Expression (Healthy Event) page, do the following:  a. For the Event ID value, type 202  b. For the Event Source value, type MyApplication  c. Click Insert.  d. Click the button to the right side of the Parameter Name box on the new entry.  e. Select Specify event specific parameter to use.  f. Specify 1 for the event parameter.  g. Click OK.  h. In the Operator box, select equals.  i. Click the button to the right side of the Value box and select ComponentName.  j. Click Finish.  9. Expand System.Health.AvailabilityState.  10. Right-click MyMP.Monitor.MyApplicationEventError and select Properties.  11. On the Health tab, for FirstEventRaised, change the Health State to Critical.  12. On the Alerting tab, check Generate alerts for this monitor.  13. Click OK. |

## See Also

[Events](#z7ddb06a4360e487fa48d5a2a615294f6)

[Event Monitors](#z9a556512ab434d75aa4dd6958c0f3457)

# How to create a repeating event monitor

The following procedure shows how to create a repeating event monitor in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 Sets the monitor to a critical state when an event in the Application event log with an event source of MyApplication and an event number of 201 is detected 3 times within 5 minutes.

 Sets the monitor to a healthy state when an event in the Application event log with an event source of MyApplication and an event number of 202 is detected.

To create a repeating event monitor

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand Computer Role 1 and then System.Health.EntityState.  3. Right-click System.Health.AvailabilityState, select New, select Windows Events, select Repeated, and then select Event Reset.  4. On the General page, do the following:  a. In the ElementID box, type MyMP.Monitor.MyApplicationRepeatingEventError.  b. In the Display Name box, type My Application Repeating Event Error.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Parent Monitor box, select System.Health.AvailabilityState.  e. In the Category box, select AvailabilityHealth. Click Next.  5. On the Repeated Event Log page, in the Log Name box, keep the default value of Application. Click Next.  6. On the Repeated Event Log Expression page, do the following:  a. For the Event ID value, type 201  b. For the Event Source value, type MyApplication  c. Click Next.  7. On the Repeated Event Detection page, do the following:  a. For the Counting Mode, select Trigger on count, sliding.  b. For the Compare Count, type 3.  c. Select Based on items occurrence within a time interval.  d. For the Interval, type 5 and select Minutes.  8. On the Repeated Event Log page, in the Log Name box, keep the default value of Application. Click Next.  9. On the Simple Event Log page, do the following:  a. For the Event ID value, type 202  b. For the Event Source value, type MyApplication  c. Click Finish.  10. Expand System.Health.AvailabilityState.  11. Right-click MyMP.Monitor.MyApplicationRepeatingEventError and select Properties.  12. On the Health tab, for RepeatedEventRaised, change the Health State to Critical.  13. On the Alerting tab, check Generate alerts for this monitor.  14. In the Alert description box, type Event 201 was detected $Data/Context/Count$ times between $Data/Context/TimeWindowStart$ and $Data/Context/TimeWindowEnd$. The first event was at $Data/Context/TimeWindowStart$. The last event was at $Data/Context/TimeWindowEnd$.  15. Click OK. |

## See Also

[Events](#z7ddb06a4360e487fa48d5a2a615294f6)

[Event Monitors](#z9a556512ab434d75aa4dd6958c0f3457)

# How to create a monitor based on a script

The following procedure shows how to create a monitor based on a monitoring script in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer with an instance of MyComputerRole1.

 Sets the monitor to a critical state when the script returns a status message of Bad.

 Sets the monitor to a healthy state when the script returns a status message of Good.

 The script accepts only a single argument for the computer name of the target object’s agent.

 The script itself is only for testing and performs no real function. It simulates a script running a synthetic transaction.

To create a script monitor

|  |
| --- |
| 1. In the Authoring Console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand Computer Role 1 and then System.Health.EntityState.  3. Right click System.Health.AvailabilityState, select New, select Scripting and then select Two State.  4. On the General page, do the following:  a. In the ElementID box, type MyMP.Monitor.MyApplicationScriptMonitor.  b. In the Display Name box, type My Application Script Monitor.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Parent Monitor box, select System.Health.AvailabilityState.  e. In the Category box, select AvailabilityHealth. Click Next.  5. On the Schedule page, in the Run every box, type 15 minutes. Click Next.  6. On the Script page, do the following:  a. For the File Name value, type MyScript.vbs  b. For the Timeout value, type 1 minutes  c. In the Script box, paste the complete contents of the following script.  sComputerName = WScript.Arguments(0)  bTestSuccessful = True  Set oAPI = CreateObject("MOM.ScriptAPI")  oAPI.LogScriptEvent "MyScript.vbs",10,4, "Running script on " & sComputerName  Set oBag = oAPI.CreatePropertyBag()  Call oBag.AddValue("ComputerName",sComputerName)  If bTestSuccessful = True Then  Call oBag.AddValue("Result","Good")  Else  Call oBag.AddValue("Result","Bad")  End If  oAPI.Return(oBag)  d. Click the Parameters button.  e. Select Target, then select (Host=Windows Computer), then select Principal Name (Windows Computer).  f. Click OK.  g. Click Next.  7. On the Unhealthy Expression page, do the following:  a. Click Insert.  b. In the Parameter Name box type Property[@Name='Result'].  c. In the Operator box select Equals.  d. In the Value box type Bad.  e. Click Next.  8. On the Healthy Expression page, do the following:  a. Click Insert.  b. In the Parameter Name box type Property[@Name='Result'].  c. In the Operator box select Equals.  d. In the Value box type Good.  e. Click Finish.  9. Expand System.Health.AvailabilityState.  10. Right click MyMP.Monitor.MyApplicationScriptMonitor and select Properties.  11. On the Health tab, for Error, change the Health State to Critical.  12. On the Alerting page, do the following:  a. Check Generate alerts for this monitor  b. In the Alert description box, type Result: $Data/Context/Property[@Name='Result']$  c. Click OK. |

## See Also

[Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8)

[Script Monitors](#zc5fdd47a651c4c488f606e54ca9c0e3c)

# How to create a WMI event monitor

The following procedure shows how to create a WMI event monitor in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

Note

In this example, criteria is included in the WMI query, so no expression is required in the monitor. The WMI event wizard in the Authoring console requires an expression for each event though. Because of this requirement, dummy expressions will be provided to complete the wizard and then deleted once the monitor is created.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 Sets the monitor to a critical state when Notepad is started on the agent computer.

 Sets the monitor to a healthy state when Notepad is ended on the agent computer.

To create a WMI event monitor

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand Computer Role 1 and then System.Health.EntityState.  3. Right-click System.Health.AvailabilityState, select New, select WMI Events, select Simple, and then select Event Reset.  4. On the General page, do the following:  a. In the ElementID box, type MyMP.Monitor.MyApplicationWMIEventError.  b. In the Display Name box, type MyApplication WMI Event Error.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Parent Monitor box, select System.Health.AvailabilityState.  e. In the Category box, select AvailabilityHealth. Click Next.  5. On the First WMI Event Provider page, do the following:  a. In the WMI Namespace box, type root\cimv2.  b. In the Query box, paste the following WMI query.  Select \* From \_\_InstanceCreationEvent WITHIN 60 Where TargetInstance ISA 'Win32\_Process' and TargetInstance.Name = 'notepad.exe'  c. In the Poll Interval box, type 60.  d. Click Next.  6. On the First Expression page, do the following:  a. Click Insert.  b. In the Parameter Name box type Dummy.  c. In the Operator box select Equals.  d. In the Value box type Dummy.  e. Click Next.  7. On the Second WMI Event Provider page, do the following:  a. In the WMI Namespace box, type root\cimv2.  b. In the Query box, paste the following WMI query.  Select \* From \_\_InstanceDeletionEvent WITHIN 60 Where TargetInstance ISA 'Win32\_Process' and TargetInstance.Name = 'notepad.exe'  c. In the Poll Interval box, type 60.  d. Click Next.  8. On the Second Expression page, do the following:  a. Click Insert.  b. In the Parameter Name box type Dummy.  c. In the Operator box select Equals.  d. In the Value box type Dummy.  e. Click Finish.  9. Expand System.Health.AvailabilityState.  10. Right-click MyMP.Monitor.MyApplicationWMIEventError and select Properties.  11. On the Configuration tab, do the following:  a. Click Configure.  b. Select the First Expression tab.  c. Click Delete.  d. Select the Second Expression tab.  e. Click Delete.  f. Click OK.  12. On the Health tab, for FirstEventRaised, change the Health State to Critical.  13. On the Alerting tab, do the following:  a. Check Generate alerts for this monitor  b. In the Alert description box, type Stopped process: $Data/Context/Collection['TargetInstance']/Property[@Name='Caption']$  c. Click OK. |

## See Also

[Events](#z7ddb06a4360e487fa48d5a2a615294f6)

[Event Monitors](#z9a556512ab434d75aa4dd6958c0f3457)

# How to create a consecutive sample performance monitor

The following procedure shows how to create a consecutive sample performance monitor in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer with an instance of MyComputerRole1.

 Monitors the memory utilization on any computer with an instance of MyComputerRole1.

 Sets the monitor to a critical state when available memory is less than 100 MB for 4 consecutive samples.

 Sets the monitor to a healthy state when available memory is above 100 MB for a single sample.

To create a consecutive sample performance monitor

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand Computer Role 1 and then System.Health.EntityState.  3. Right click System.Health.PerformanceState, select New, select Windows Performance, select Static Thresholds, and then select Consecutive Samples.  4. On the General page, do the following:  a. In the ElementID box, type MyMP.Monitor.MyApplicationConsecutiveSampleMonitor.  b. In the Display Name box, type MyApplication Consecutive Sample Monitor.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Parent Monitor box, select System.Health.PerformanceState.  e. In the Category box, select PerformanceHealth. Click Next.  5. On the Performance Counter page, do the following:  a. Click Select.  b. In the Object box, select Memory.  c. In the Select counter from list box, select Available MBytes.  d. Click OK.  e. In the Interval box, type 15 and select Minutes.  f. Click Next.  6. On the Threshold Comparison page, do the following:  a. In the Value box, select less than and type 100.  b. In the Number of samples box type 4.  c. Click Finish.  7. Expand System.Health.PerformanceState.  8. Right-click MyMP.Monitor.MyApplicationConsecutiveSampleMonitor and select Properties.  9. On the Health tab, for Condition True, change the Health State to Critical.  10. On the Alerting tab, Check Generate alerts for this monitor.  11. Click OK. |

## See Also

[Performance Data](#z336bae3fc9794e7ca162c626c55e6d94)

[Performance Monitors](#z7335542b6b38488aa703e6eaae902a46)

# How to create a dependency monitor

The following procedures show how to create a dependency monitor in the Operations Manager 2007 Authoring console. Before you perform these procedures, you must complete the following prerequisite procedures:

 [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) - Create classes that will serve as both the source and target in relationships.

 [How to Create a Containment Relationship](#z8d524c60568f4c37a7ed8c9e86515c32) – Create containment relationship between classes that the dependency monitor will use.

The dependency monitors created in this procedure has the following characteristics:

 Causes the health of instances of MyComputerRole1 to roll up to instances of MyComputerRole2 on the same agent.

 One dependency monitor rolls up the health of the Availability aggregate monitor of MyComputerRole1 to the Availability aggregate monitor of MyComputerRole2.

 One dependency monitor rolls up the health of the Performance aggregate monitor of MyComputerRole1 to the Performance aggregate monitor of MyComputerRole2.

 Performs health rollup using worst state policy.

To create a dependency monitor on Availability aggregate monitor

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand MyMP.MyComputerRole2 and then System.Health.EntityState.  3. Right-click System.Health.AvailabilityState, select New, and then select Dependency Monitor.  4. In the Choose a unique identifier box, type MyMP.MyComputerRole2DependsOnMyComputerRole1Availability. Click OK.  5. On the General tab, do the following:  a. In the Display Name box, type My ComputerRole 2 Depends On My Computer Role 1 Availability.  b. In the Target box, select MyMP.MyComputerRole2.  c. In the Parent Monitor box, select System.Health.AvailabilityState.  6. On the Monitor Dependency tab, do the following:  a. Expand Entity Health for My Computer Role 1.  b. Select Availability.  7. On the Health Rollup Policy page, select Worst state of any member.  8. Click OK. |

To create a dependency monitor on Performance aggregate monitor

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand MyMP.MyComputerRole2 and then System.Health.EntityState.  3. Right-click System.Health.PerformanceState, select New, and then select Dependency Monitor.  4. In the Choose a unique identifier box, type MyMP.MyComputerRole2DependsOnMyComputerRole1Performance. Click OK.  5. On the General tab, do the following:  a. In the Display Name box, type My ComputerRole 2 Depends On My Computer Role 1 Performance.  b. In the Target box, select MyMP.MyComputerRole2.  c. In the Parent Monitor box, select System.Health.PerformanceState.  6. On the Monitor Dependency tab, do the following:  a. Expand Entity Health for My Computer Role 1.  b. Select Performance.  7. On the Health Rollup Policy page, select Worst state of any member.  8. Click OK. |

## See Also

[Dependency Monitors](#zfee2d08685ed40439fe1f8df64226740)

# Creating Diagnostics and Recoveries

The procedures in this section provide guidance on how to create diagnostics and recoveries in the Operations Manager 2007 Authoring console.

## In This Section

[How to create a diagnostic](#z629f8f58fa52405480b54f767f0b61b7)

|  |
| --- |
| Create a diagnostic that runs a command. |

[How to create a recovery based on the output of a diagnostic](#zdf80cc198a3f4627ad487b5cfa8e4c86)

|  |
| --- |
| Create a diagnostic and recovery that run a script. The recovery only runs if certain output from the diagnostic is received. |

# How to create a diagnostic

The following procedure shows how to create a diagnostic that runs a command in the Operations Manager 2007 Authoring console. Before you perform these procedures, you must complete the following prerequisite procedures:

 [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) Create the target class for the monitor.

 [How to create an event monitor](#za74f6dde42c8497aab3ece252496f4aa) Create the monitor that the diagnostic will respond to.

The diagnostic created in this procedure has the following characteristics:

 Automatically runs when the MyMP.Monitor.MyApplicationEventError monitor goes to a critical state.

 Runs tasklist.exe to return a list of running processes on the agent computer.

To create a diagnostic that runs a command

|  |
| --- |
| 1. In the Authoring Console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand MyMP.MyComputerRole1, and then expand System.Health.EntityState, and then expand System.Health.AvailabilityState.  3. Right-click MyMP.Monitor.MyApplicationEventError and select Properties.  4. Select the Diagnostic and Recovery tab.  5. Click Add and then Diagnostic for a critical health state  6. In the Choose a unique identifier box, type MyMP.MyApplicationEventError.CommandDiagnostic and then click OK.  7. On the General tab, do the following:  a. In the Name box, type Check Running Processes.  b. In the Target box, make sure that MyMP.MyComputerRole1 is selected.  8. On the Configuration tab, do the following:  a. In the Execute on monitor box, make sure that MyMP.Monitor.MyApplicationEventError is selected.  b. In the Execute when monitor’s health is box, make sure that Error is selected.  9. On the Modules tab, do the following:  a. Under Actions, click Create.  b. In the Choose Module Type box, select System.CommandExecutorProbe.  c. In the Module ID box, type Command. Click OK.  d. Click Edit.  e. Click Configure.  f. In the Full path to file box, type %windir%\system32\tasklist.exe.  g. Click OK.  h. Click OK.  i. Click OK.  j. Click OK. |

## See Also

[Diagnostics and Recoveries](#z6d472fd061a64c7ca44f19ddb8e3cd6e)

# How to create a recovery based on the output of a diagnostic

The following procedure shows how to create a recovery in the Operations Manager 2007 Authoring console that runs after a diagnostic. The recovery only runs if the output from the diagnostic meets certain criteria. Before you perform these procedures, you must complete the following prerequisite procedures.

 [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) Create the target class for the monitor.

 [How to create an event monitor](#za74f6dde42c8497aab3ece252496f4aa) Create the monitor that the diagnostic will respond to.

The diagnostic created in this procedure has the following characteristics:

 Automatically runs when the MyMP.Monitor.MyApplicationEventError monitor goes to a critical state.

 Runs a script that returns a single value in a property bag.

 The script itself is only for testing and performs no real function. This script only simulates a script that gathers diagnostic data and returns that data in a property bag.

The recovery created in this procedure has the following characteristics:

 Runs only when the value of the Result property in the property bag from the diagnostic is Positive.

 Runs a script that writes an event to the OperationsManager event log on the agent and returns a message to the Operations console.

 The value of the Result property in the property bag from the diagnostic is sent into the recovery script through an argument. This value is used in the event and the message created by the script.

 The script itself is only for testing and performs no real function. This script only simulates a script that performs a recovery action and returns the results that are shown in an event and to the Operations console.

To create a diagnostic that runs a script

|  |
| --- |
| 1. In the Authoring Console, select Health Model, and then select Monitors.  2. In the Monitors pane, expand MyMP.MyComputerRole1, expand System.Health.EntityState, and then expand System.Health.AvailabilityState.  3. Right-click MyMP.Monitor.MyApplicationEventError and select Properties.  4. Select the Diagnostic and Recovery tab.  5. Under Configure diagnostic tasks, click Add and then Diagnostic for a critical health state  6. In the Choose a unique identifier box, type MyMP.MyApplicationEventError.ScriptDiagnostic and then click OK.  7. On the General tab, do the following:  a. In the Name box, type Run diagnostic script.  b. In the Target box, make sure that MyMP.MyComputerRole1 is selected.  8. On the Configuration tab, do the following:  a. In the Execute on monitor box, make sure that MyMP.Monitor.MyApplicationEventError is selected.  b. In the Execute when monitor’s health is box, make sure that Error is selected.  9. On the Modules tab, do the following:  a. Under Actions, click Create.  b. In the Choose Module Type box, select Microsoft.Windows.ScriptPropertyBagProbe.  c. In the Module ID box, type Script. Click OK.  d. Under Actions, click Edit.  e. In the ScriptName box, type MyDiagnosticScript.vbs.  f. In the Arguments box, clear the existing text.  g. In the TimeoutSeconds box, type 300.  h. Click Edit. This starts the external editor.  i. Paste the contents of the following script between the ScriptBody tags in the XML. Replace any text that might already exist.  bTestSuccessful = True  Set oAPI = CreateObject("MOM.ScriptAPI")  Set oBag = oAPI.CreatePropertyBag()  If bTestSuccessful = True Then  Call oBag.AddValue("Result","Positive")  Else  Call oBag.AddValue("Result","Negative")  End If  oAPI.Return(oBag)  j. Close the external editor to save the script back to the module.  k. Click OK.  l. Click OK. |

To create a recovery based on the output of a diagnostic

|  |
| --- |
| 1. Under Configure recovery tasks, click Add and then Recovery for a critical health state  2. In the Choose a unique identifier box, type MyMP.MyApplicationEventError.ScriptRecovery and then click OK.  3. On the General tab, do the following:  a. In the Name box, type Run recovery script.  b. In the Target box, make sure that MyMP.MyComputerRole1 is selected.  4. On the Configuration tab, do the following:  a. In the Execute on monitor box, make sure that MyMP.Monitor.MyApplicationEventError is selected.  b. Select Execute after diagnostic.  c. Select MyMP.MyApplicationEventError.ScriptDiagnostic.  5. On the Modules tab, do the following:  a. Under Condition Detection, click Create.  b. In the Choose Module Type box, select System.ExpressionFilter.  c. In the Module ID box, type FilterDiagnostic. Click OK.  d. Under Condition Detection, click Edit.  e. Click Configure.  f. Click Insert.  g. In the Parameter column, type Diagnostic/DataItem/Property[@Name='Result'].  h. In the Operator column, select Equals.  i. In the Parameter column, type Positive.  j. Click OK.  k. Click OK.  l. Under Actions, click Create.  m. In the Choose Module Type box, select Microsoft.Windows.ScriptWriteAction.  n. In the Module ID box, type Script, and then click OK.  o. Click Edit.  p. Click Configure.  q. In the File Name box, type MyRecoveryScript.vbs.  r. Paste the contents of the following script in the Script box.  sDiagnosticOutput = WScript.Arguments(0)  sMessage = "Recovery ran after diagnostic result of " & sDiagnosticOutput & "."  Set oAPI = CreateObject("MOM.ScriptAPI")  oAPI.LogScriptEvent "MyRecoveryScript.vbs",100,4,sMessage  s. Click Parameters.  t. In the Parameters box, type $Data/Diagnostic/DataItem/Property[@Name='Result']$.  u. Click OK.  v. Click OK.  w. Click OK.  x. Click OK.  y. Click OK. |

## See Also

[Diagnostics and Recoveries](#z6d472fd061a64c7ca44f19ddb8e3cd6e)

# Creating Rules

The following procedures provide guidance on how to create rules in the Operations Manager 2007 Authoring console.

## In This Section

[How to create a delimited text log alerting rule](#z41e51e46165a402e9d4bcb435eefbd06)

|  |
| --- |
| Create a rule that creates an alert from a delimited text log. |

[How to create a WMI performance collection rule](#z8c6fcb7fb868472ab90f1e199ef6d192)

|  |
| --- |
| Create a rule that collects performance data from a WMI query. |

[How to create a script-based performance collection rule](#z860f237a580b41d297e54410d8d7cf67)

|  |
| --- |
| Create a performance collection rule using data from a script. |

[How to create a script-based event collection rule](#z7a3d883db9404b608b162199160b0d15)

|  |
| --- |
| Create an event collection rule using data from a script. |

# How to create a delimited text log alerting rule

The following procedure shows how to create an alert rule from a delimited text log in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 Watches a log file that has a naming pattern of MyApp\*.log located in the c:\logs directory. The file is expected to be comma delimited.

 Creates an alert with a critical state when the string “error” is found in the second field.

 Includes the first, third, and fourth fields in the description of the alert.

 Suppresses alerts when the name of the logging computer and the value in the first field match.

To create a delimited text log alert rule

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Rules.  2. Right-click in the Rules pane, select New, select Alerting, and then select Text Log (Delimited).  3. On the General page, do the following:  a. In the ElementID box, type MyMP.Rule.AlertOnDelimitedTextLog.  b. In the Display Name box, type MyApplication Delimited Log Error.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Category box, select Alert. Click Next.  4. On the Application Log Data Source page, do the following:  a. In the Directory box, type c:\logs.  b. In the Pattern box, type MyApp\*.log.  c. In the Separator box, type a COMMA. Click Next.  5. On the Build Event Expression page, do the following:  a. Click Insert.  b. In the Parameter Name box type Params/Param[2].  c. In the Operator box select Contains.  d. In the Value box type error.  e. Click Next.  6. On the Configure Alerts page, do the following:  a. In the Alert name box, type Error found in MyApplication delimited text log..  b. Click the button to the right side of the Alert description box.  c. Clear the text in the Value box.  d. Select Data, then Params, then Param.  e. Replace the text <<INT>> with 1.  f. Move to the end of the line and press the ENTER key.  g. Select Data, then Params, then Param.  h. Replace the text <<INT>> with 3.  i. Move to the end of the line and press the ENTER key.  j. Select Data, then Params, then Param.  k. Replace the text <<INT>> with 4.  l. Move to the end of the line and press the ENTER key.  m. Click OK.  7. Click Finish.  8. Right-click MyMP.Rule.AlertOnDelimitedTextLog and select Properties.  9. On the Modules tab, do the following:  a. Click the Edit button next to the Action pane.  b. Click the Configure button.  c. Click the Alert Suppression button.  d. Select Logging Computer and Parameter 1.  10. Click OK.  11. Click OK.  12. Click OK.  13. Click OK. |

## See Also

[Events](#z7ddb06a4360e487fa48d5a2a615294f6)

[Alert Rules](#z0ee45649328c4c039a427258b9e3edce)

# How to create a WMI performance collection rule

The following procedure shows how to create a WMI performance collection rule in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 Collects the size for a file that is named C:\Logs\MyAppFile.txt.

To create a WMI performance collection rule

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Rules.  2. Right-click in the Rules pane, select New, select Collection, select Performance Based, and then select WMI Performance Collection.  3. On the General page, do the following:  a. In the ElementID box, type MyMP.Rule.CollectWMIPerformance.  b. In the Display Name box, type My Application Collect WMI Performance.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Category box, select PerformanceCollection. Click Next.  4. On the WMI Configuration page, do the following:  a. In the WMI Namespace box, type root\cimv2.  b. In the Query box, paste the following WMI query.  Select Name,FileSize From CIM\_DataFile Where Name = 'C:\\Logs\\MyAppFile.txt'  c. In the Query Interval box, type 900.  d. Click Next.  5. On the Performance Mapper page, do the following:  a. In the Object box, type MyApp.  b. In the Counter box, type FileSize.  c. In the Instance box, type $Data/Property[@Name=’Name’]$.  d. In the Value box, type $Data/Property[@Name=’FileSize’]$.  e. Click Finish. |

## See Also

[Performance Data](#z336bae3fc9794e7ca162c626c55e6d94)

[Performance Collection Rules](#zceadc507cc0e4de39dac1bb6f3da3b52)

# How to create a script-based performance collection rule

The following procedure shows how to create a performance collection rule by using a monitoring script in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 The script accepts two parameters, one for computer name and another for the version of the application that is stored as a property on the target class.

 The script itself is only for testing and performs no real function. It simulates a script running a synthetic transaction and returning a property bag with static values.

To create a script based performance collection rule

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Rules.  2. Right-click in the Rules pane, select New, select Collection, select Performance Based, and then select Script Based Performance Collection.  3. On the General page, do the following:  a. In the ElementID box, type MyMP.Rule.CollectScriptPerformance.  b. In the Display Name box, type My Application Collect Script Performance.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Category box, select PerformanceCollection. Click Next.  4. On the Schedule page, in the Run every box, type 15 minutes.  5. On the Script page, do the following:  a. For the File Name value, type MyPerfCollectionScript.vbs  b. For the Timeout value, type 1 minutes  c. In the Script box, paste the complete contents of the following script.  sComputerName = WScript.Arguments(0)  sVersion = WScript.Arguments(1)  Set oAPI = CreateObject("MOM.ScriptAPI")  Set oBag = oAPI.CreatePropertyBag()  Call oBag.AddValue("ComputerName",sComputerName)  Call oBag.AddValue("InstanceName","MyInstance")  Call oBag.AddValue("Value",10)  oAPI.Return(oBag)  d. Click the Parameters button.  e. Select Target, select (Host=Windows Computer), and then select Principal Name (Windows Computer).  f. Type a SPACE.  g. Select Target and then Version (My Computer Role Base).  h. Click OK.  i. Click Next.  6. On the Performance Mapper page, do the following:  a. In the Object box type MyApp.  b. In the Counter box type MyCounter.  c. In the Instance box type $Data/Property[@Name=’InstanceName’]$.  d. In the Value box type $Data/Property[@Name=’Value’]$.  e. Click Finish. |

## See Also

[Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8)

[Performance Collection Rules](#zceadc507cc0e4de39dac1bb6f3da3b52)

# How to create a script-based event collection rule

The following procedure shows how to create an event collection rule by using a monitoring script in the Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the target class.

The monitor created in this procedure has the following characteristics:

 Runs on any computer that has an instance of MyComputerRole1.

 The script accepts two parameters, one for computer name and another for the version of the application that is stored as a property on the target class.

 The script itself is only for testing and performs no real function. It simulates a script running a synthetic transaction and returning a property bag with static values.

To create a script based event collection rule

|  |
| --- |
| 1. In the Authoring console, select Health Model, and then select Rules.  2. Right-click in the Rules pane, select New, select Collection, select Event Based, and then select Script Based Event Collection.  3. On the General page, do the following:  a. In the ElementID box, type MyMP.Rule.CollectScriptEvent.  b. In the Display Name box, type My Application Collect Script Event.  c. In the Target box, select MyMP.MyComputerRole1.  d. In the Category box, select EventCollection. Click Next.  4. On the Schedule page, in the Run every box, type 15 minutes.  5. On the Script page, do the following:  a. For the File Name value, type MyEventCollectionScript.vbs  b. For the Timeout value, type 1 minutes  c. In the Script box, paste the complete contents of the following script.  sComputerName = WScript.Arguments(0)  sVersion = WScript.Arguments(1)  Set oAPI = CreateObject("MOM.ScriptAPI")  Set oBag = oAPI.CreatePropertyBag()  Call oBag.AddValue("ComputerName",sComputerName)  Call oBag.AddValue("EventID",100)  Call oBag.AddValue("Param1",”ParamValue”)  oAPI.Return(oBag)  d. Click Parameters.  e. Select Target, select (Host=Windows Computer), and then select Principal Name (Windows Computer).  f. Type a SPACE.  g. Select Target and then Version (My Computer Role Base).  h. Click OK.  i. Click Next.  6. On the Performance Mapper page, do the following:  a. In the Computer box type $Data/Property[@Name='ComputerName']$.  b. In the Event source box type MyApp.  c. In the Event log box type CustomScript.  d. In the Event ID box type $Data/Property[@Name='EventID']$.  e. In the Category box type 0.  f. In the Level box select Information.  g. Click the Parameters button.  h. Type $Data/Property[@Name='ParamValue']$  i. Click OK.  j. Click Finish. |

## See Also

[Monitoring Scripts](#zbd29182d0f4f46c7bbcab5e60e5117d8)

[Event Collection Rules](#z00cda1344d3e46bcaa4d67045a50702e)

# Presentation

The topics in this section provide background information and recommendations on effective methods for presenting data and guidance to the user of the management pack. The following sections are not yet complete. When they are completed, they will describe the referenced topics in detail.

## In This Section

[Views](#z92fc07ac46414ed3830fa84595b84557)

|  |
| --- |
| Describes the different types of views in the Operations Console and provides best practices and processes for including them in a management pack. |

[Product Knowledge](#z2011b2b8e5bb41308140f09b60c46ac7)

|  |
| --- |
| Provides best practices for including product knowledge in a management pack. |

# Views

Views provide access to data that is collected in the Operations Manager database. Management packs will typically include multiple views to provide access to its classes and collected operations data. Multiple views in the management pack may provide access to same data but presented in different formats relevant to different monitoring scenarios.

The following views are available in Operations Manager 2007:

|  |  |
| --- | --- |
| Alert | Listing of alerts matching specific criteria. |
| Events | Listing of events matching specific criteria. |
| State | Listing of instances of a particular class in the management pack and their current health state. |
| Performance | Graph of performance data matching specific criteria. |
| Diagram | Graphical representation of instances of specified classes and their current health state. |
| Task Status | Listing of events related to the status of running tasks. |
| Web Page | Link to an external Web page. |
| Dashboard | Multiple related views in a single screen. |

When this section is completed, it will include a complete discussion of the different kinds of views, provide recommendations for designing them, and provide instructions on building them in a management pack.

# Product Knowledge

Product Knowledge is static information that is added to management pack elements that the user can access in the Operations console. Following are examples of activities that the information may assist the user with:

 Diagnosing the potential causes for a particular alert

 Understanding the details of how a monitor operates

 Recommending how to use a task.

Adding product knowledge to a management pack can help the user in the daily operations by giving them detailed knowledge about different elements in the management pack as the need for this information occurs.

When this section is completed, it will provide recommendations on the kinds of information to include in product knowledge and instructions for adding it to existing management pack elements.

# Composition

The Operations Manager 2007 authoring console allows a variety of different monitoring scenarios to be implemented by using wizards that do not require extensive knowledge of how these scenarios are actually implemented in the management pack.

## In This Section

[Key Concepts](#z437969d9ecc44f69a5257d348f3a3bd2)

|  |
| --- |
| Introduces the concepts of modules and workflows and identifies the different types of each. |

[Designing Custom Modules and Workflows](#z1c2f0854062c44ce81f98a00308763f0)

|  |
| --- |
| Explains the criteria that you must consider for designing different types of custom modules and workflows. |

[Creating Custom Modules and Workflows](#z095261514a654e0b87cc467808cdfc10)

|  |
| --- |
| Demonstrates how to create custom modules and workflows for implementing different scenarios. |

# Key Concepts

This section provides high-level background information on the operation of modules and workflows in Operations Manager 2007.

## In This Section

[Module and Workflow Basics](#z3bff4c599e23454c8d649f15fe0ed38c)

|  |
| --- |
| Describes the basic operation of modules and workflows. |

[Modules Types](#zcb0b7c2586ad4ff5b6eea556475e6881)

|  |
| --- |
| Provides the details of the different types of modules. |

[Module Implementations](#zb3f7fae7291c49a9b153ef4cc51370aa)

|  |
| --- |
| Defines the different module implementations with particular detail for composite modules. |

[Kinds of Workflows](#za67d726a39e44e2c9ff96e704448c6ca)

|  |
| --- |
| Provides details of the different kinds of workflows. |

[Cookdown](#z1f8fb91b1e814c8daacf758edb2f3141)

|  |
| --- |
| Describes the concept of cookdown and how custom modules and workflows can be designed to take advantage of it for more efficient operation. |

# Module and Workflow Basics

A workflow in Operations Manager 2007 refers to a running process that is defined in a management pack and run by the Operations Manager agent. Examples of different kinds of workflows are monitors, rules, and discoveries. The primary job of the Operations Manager agent is to load and run workflows defined by the management packs relevant to the applications installed on the agent.

Workflows are composed of one or more modules. A module in Operations Manager performs a discrete function based on information that is provided to the module through parameters. For example, a module may detect an event, collects a particular performance counter, runs a script, or generates an alert. A workflow can achieve complex functionality by piecing together different modules that perform different required actions.

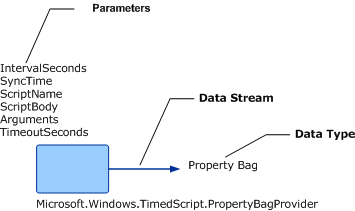
Wizards in the Authoring console let workflows such as monitors and rules be created without direct knowledge of the modules that they will use. These wizards address a limited set of specific scenarios, but with knowledge of the underlying structures, custom workflows can be made up from any valid combination of available modules to achieve specific functionality.

## Modules

Management pack libraries contain many modules that are individually documented in the Module Types Reference section of the Operations Manager 2007 Management Pack Development Kit (MPDK). These modules may be used to compose custom workflows or to create custom modules required for specific functionality in a management pack.

A sample module is illustrated in the following diagram. The module is included in the Microsoft Windows Library management pack and has the name Microsoft.Windows.TimedScript.PropertyBagProvider. The function of this module is to run a script according to a specified schedule and return the output in the form of a property bag.

Module



The module can be inspected by opening the Windows Core Library management pack in the Authoring console. Detailed documentation for the module is provided in the MPDK under Microsoft.Windows.ScriptProbeAction. The reason that the module does not have its own topic is that it and other modules make use of the Microsoft.Windows.ScriptProbeAction module. Instead of providing separate documentation for each module, the MPDK groups similar modules under a common topic.

To perform its required function, the module requires a set of parameters that specify such information as the script itself and how frequently the script should be run. Any output from the script is formatted as a particular data type and sent into the data stream, where the output is retrieved by the next module in the workflow. These concepts are detailed in the following sections.

### Module Parameters

Most modules will require one or more parameters. These parameters are values that the module requires in order to perform its required functionality. Explicit values may be provided for these parameters, or they may be $Target or $Data variables that are determined at run time. A parameter can be defined to be required or optional. Any workflow or composite module that uses a module must provide values for each of its required parameters.

The following table lists the parameters for the previous sample module:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter Name | Description | Required? | Possible Values |
| IntervalSeconds | How frequently the script should run. | Yes | Typically explicit value, such as 900 seconds. |
| SyncTime | Synchronization time for scheduling of the script. | No | Typically explicit value, such as 00:15, or blank if synchronization is not required. |
| ScriptName | Name of the script. | Yes | Explicit value, such as MyScript.vbs. |
| ScriptBody | Body of the script. This is the full text of the script itself. | Yes | Explicit value with the complete body of the script. |
| Arguments | Values for arguments that the script requires. | No | Combination of explicit values and $Target variables providing the command-line arguments sent to the script. |
| TimeoutSeconds | Number of seconds that the script can run before it is automatically stopped. | Yes | Explicit value, such as 300 seconds. |

The list of parameters for a particular module is available by viewing its management pack in the Operations console or an XML editor. When a module is used in the Authoring console, its list of required parameters will be automatically populated. If optional parameters are used, they must be added manually.

### Data Stream

Modules in a workflow exchange data through a data stream. Data sent from one module is accepted by the next module in the workflow. The receiving module is expected to perform some function by using the received data and then potentially sending its own data as output where it may be accepted by another module.

Each workflow has its own data stream that is created when the workflow runs. If there are multiple copies of a particular workflow running on an agent, each will have its own data stream.

### Data Types

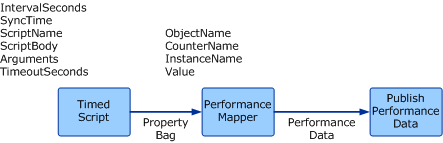
Data output from a module will match a particular data type, and modules that accept data will require a particular type of data for input. When composing a workflow, you must make sure that the output data type of one module matches the required input data type of the next module. If they do not match, another module can be used to translate from one type to another.

The previous example module above sends output data in a property bag. Any module receiving this data in a workflow would be required to accept property bag data as input. If a module requiring another data type were used, for example, performance data, another module would be required in the workflow to map the property bag to the required data type.

## Workflows

The following diagram shows a sample workflow. This particular sample represents a rule that runs a script and collects performance data in the Operations Manager database. It uses three modules to perform this function. The following diagram shows the parameters that are required for each module and the data type for the output and input of each.

Workflow



The first module is Microsoft.Windows.TimedScript.PropertyBagProvider that was introduced in the previous module example. This module requires values for the listed parameters, which include how frequently the script should run, the name and body of the script, and a set of arguments that will be sent to the script. The values for these parameters would be provided by the configuration of the workflow itself.

The final module Microsoft.SystemCenter.CollectPerformanceData accepts performance data from the data stream and stores it in the Operations Manager database. Because the output from the first module is a property bag, a separate module is required to map data from a property bag to performance data.

The second module in the workflow System.Performance.DataGenericMapper provides the required mapping between data types. This module can accept any input data type and output performance data. In order to perform this function, it requires the parameters that are shown in the diagram. The object name and counter name would presumably be static values, although the instance, if provided, would probably come from a value in the property bag or from a $Target variable using the value of a property on the target object to uniquely identify the instance. The value for the data would most likely come from a value from the property bag that was calculated during execution of the script.

The final module Microsoft.SystemCenter.CollectPerformanceData requires no parameters. As long as the data is provided to the module from the data stream, the module has all the information that it requires to save that data to the Operations Manager database.

### Workflow Targets

Every workflow requires a target class which defines the agents the workflow will run on and how many copies of the workflow will be run.

A separate copy of a workflow is loaded by an agent for each instance of the target class managed by the agent. If an agent has no instances of the target class, the workflow is not loaded. If the agent has multiple instances of the target class, multiple copies of the workflow are loaded. Each copy of the workflow runs independently of the others and will have its own data stream and its own set of property values.

Explicit values that are provided for parameters that are passed to modules in the workflow are common for all copies of the workflows. $Target variables will resolve to different values depending on the values of the properties for the individual target instances.

# Modules Types

Each module in an Operations Manager 2007 management pack will be one of four specific types as detailed in the following sections. Understanding the definition of each type and their role in different workflows is critical to designing and creating custom monitoring scenarios.

## In This Section

[Data Source Modules](#zf8a30e4714714025814e86f7ca921bed)

|  |
| --- |
| Collect information at the start of a workflow and send that information into the data stream. |

[Probe Action Modules](#z725ac7d5c80842dc8464ff89d345aa17)

|  |
| --- |
| Collect information in response to a trigger and send that information into the data stream. |

[Condition Detection Modules](#z4516fb75904f4b2fa00c1d533e06501a)

|  |
| --- |
| Filter data, map data to an alternate type, or consolidate multiple data streams. |

[Write Action Modules](#ze0df77a6eb2141a68c9a99d8c6855634)

|  |
| --- |
| Accept a single input from the data stream and make a change to system state. |

# Data Source Modules

Data source modules in Operations Manager 2007 accept no input from the data stream because they are used at the start of a workflow. They produce a single output data stream. Data source modules are used in rules, discoveries, and regular detection in monitor types. They are always the first module in these kinds of workflows, since they define the schedule for the operation of the workflow and the initial data that is retrieved.

Data source modules collect information from an external source and pass that information into the data stream. They start workflows by using a timed-trigger mechanism that does not require user intervention. Data source modules perform activities such as collecting events from a specified log, sampling performance data at set intervals, or running a script according to a defined schedule.

Data source module



Data source modules are not expected to make any changes to the environment but instead collect information that is sent into the data stream for other modules to potentially work with. This restriction cannot be enforced with custom modules performing actions such as running a script. The management pack author is expected to make sure that any such scripts or other actions used in a data source do not change the managed system.

## Common Data Source Modules

The following table lists typically used data source modules. This is not a complete list of all data source modules that are available, but they are the most common modules that management pack authors will directly use in building custom workflows. The Module Types Reference can be accessed for a complete list of data source modules that are available to use for composing custom modules and workflows.

|  |  |  |  |
| --- | --- | --- | --- |
| Module Name | Management Pack | Function | Output Data Type |
| Microsoft.Windows.Discovery.WMISinglePropertyProvider | Microsoft.Windows.Library | Runs a WMI query and returns discovery data. Includes an expression to determine whether discovery data should be created. | System.Discovery.Data |
| Microsoft.Windows.EventProvider | Microsoft.Windows.Library | Retrieves events with specified criteria from the Windows event log. | Microsoft.Windows.EventData |
| Microsoft.Windows.FilteredRegistryDiscoveryProvider | Microsoft.Windows.Library | Collects keys and values from the registry and returns discovery data. Includes an expression to determine whether discovery data should be created. | System.Discovery.Data |
| Microsoft.Windows. RegistryDiscoveryProvider | Microsoft.Windows.Library | Collects keys and values from the registry and returns discovery data. | System.Discovery.Data |
| Microsoft.Windows.TimedPowerShell.DiscoveryProvider | Microsoft.Windows.Library | Runs a Windows PowerShell script at regular intervals and returns output as discovery data. | System.Discovery.Data |
| Microsoft.Windows.TimedScript.DiscoveryProvider | Microsoft.Windows.Library | Runs a script at set intervals and returns output as discovery data. | System.Discovery.Data |
| Microsoft.Windows.TimedScript.EventProvider | Microsoft.Windows.Library | Runs a script at set intervals and maps property bag output to event. | System.Event.Data |
| Microsoft.Windows.TimedScript.PerformanceProvider | Microsoft.Windows.Library | Runs a script at set intervals and maps property bag output to performance data. | System.Performance.Data |
| Microsoft.Windows.TimedScript.PropertyBagProvider | Microsoft.Windows.Library | Runs a script at set intervals and returns output as property bag. | System.PropertyBagData |
| Microsoft.Windows.WmiEventProvider | Microsoft.Windows.Library | Monitors for occurrence of specified WMI event. | System.PropertyBagData |
| System.Discovery.Scheduler | System.Library | Output a trigger on a schedule. Used only for discoveries. | SystemTriggerData |
| System.Performance.DataProvider | System.Performance.Library | Collects a performance counter at set intervals. | System.Performance.Data |
| System.Performance.OptimizedDataProvider | System.Performance.Library | Collects a performance counter at set intervals only if delta between samples matches specified criteria. | System.Performance.Data |
| System.Scheduler | System.Library | Output a trigger on a schedule. | System.TriggerData |
| System.SimpleScheduler | System.Library | Output a trigger on a schedule using frequency and sync time only. | System.TriggerData |

# Probe Action Modules

Probe action modules resemble data source modules because they collect information and pass it into the data stream. A probe action module is also not intended to change the agent in any way. The difference is that a probe action requires some trigger to start it.

Probe action module



Probe action modules accept a single input from the data stream and produce a single output data stream. They are directly used in monitor types for On Demand Detection and can be used in Diagnostics and Recoveries. All those workflow types provide a trigger that is required to initiate the probe action. Probe action modules are most frequently used within composite data source modules. Many of the data source modules that are included in the management pack libraries actually contain probe action modules to perform their core functionality.

## Common Probe Action Modules

The following table lists common probe action modules. This is not a complete list of all probe action modules that are available. However, they are the most common probe action modules that management pack authors will directly use in building custom workflows and modules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Module Name | Management Pack | Function | Input Data Type | Output Data Type |
| Microsoft.Windows.PowerShellDiscoveryProbe | Microsoft.Windows.Library | Runs a Windows PowerShell discovery script. | System.BaseData | System.DiscoveryData |
| Microsoft.Windows.PowerShellPropertyBagProbe | Microsoft.Windows.Library | Runs a Windows PowerShell monitoring script. Requires input data. | System.BaseData | System.DiscoveryData |
| Microsoft.Windows.PowerShellPropertyBagTriggerOnlyProbe | Microsoft.Windows.Library | Runs a Windows PowerShell monitoring script. Does not require input data. | None | System.DiscoveryData |
| Microsoft.Windows.ScriptPropertyBagProbe | Microsoft.Windows.Library | Runs a monitoring script. | System.BaseData | System.PropertyBagData |
| Microsoft.Windows.WmiProbe | System.Library | Runs WMI query to retrieve data. Requires input data. | System.BaseData | System.PropertyBagData |
| Microsoft.Windows.WmiTriggerProbe | Microsoft.Windows.Library | Runs WMI query to retrieve data. Does not require input data. | None | System.PropertyBagData |
| System.CommandExecuterProbe | System.Library | Run a command line program. | System.BaseData | System.CommandOutput |
| System.PassThroughProbe | System.Library | Use to provide input to non-trigger probe action modules for on demand monitor type workflows. | None | System.BaseData |

# Condition Detection Modules

Condition detection modules in an Operations Manager 2007 management pack take one or more inputs from the data stream and produce a single output. They are used regularly in rules and monitor types. Although they are not directly used in other workflow types, they are included in composite modules whenever their functionality is required.

Condition detection modules perform one of the following three functions:

 Filter the input data according to a specified condition.

For example, a condition detection module might be used to determine whether a particular sampled performance counter exceeded a specified threshold or whether an event matched certain criteria.

 Map data from one data type to another.

For example, condition detection might be used to map data from a property bag generated by a script run from a data source module to performance data required for insertion into the Operations Manager database.

 Consolidate the data from multiple data streams.

For example, a particular workflow might only generate a message when multiple events of the same type are generated in a specified time window. A condition detection module can be used to compile the events and determine when the specified number of events have been collected.

Condition detection module



## Common Condition Detection Modules

The following table lists frequently used condition detection modules. This is not a complete list of all condition detection modules that are available, but they are the most common ones that management pack authors will directly use in building custom workflows.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Module Name | Library | Function | Input Data Type | Output Data Type |
| System.Event.GenericDataMapper | System.Library | Maps data to an event | System.BaseData | System.Event.Data |
| System.ExpressionFilter | System.Library | Evaluates data according to specified criteria to determine whether the data should be allowed through to the next module in the workflow | System.BaseData | System.BaseData |
| System.OptimizedCollectionFilter | System.Performance.Library | Optimizes performance data | System.Performance.Data | System.Performance.Data |
| System.Performance.DataGenericMapper | System.Performance.Library | Maps data to performance data. | System.BaseData | System.Performance.Data |

# Write Action Modules

Write action modules accept a single input from the data stream and make a change to system state. They have no output into the data stream and represent the end of a workflow. The change made by a write action module could be to the monitored system or in Operations Manager itself. Examples include generating an alert, writing information to the Operations Manager database, or running a script to try to fix a detected problem in an application.

Write action module



## Common Write Action Modules

The following table lists common write action modules. This is not a complete list of all write action modules that are available. However, they are the most common ones that management pack authors will directly use in building custom workflows.

|  |  |  |  |
| --- | --- | --- | --- |
| Module Name | Library | Function | Input Data Type |
| Microsoft.SystemCenter.CollectEvent | Microsoft.SystemCenter.Library | Write events into the Operations Manager database. | System.Event.Data |
| Microsoft.SystemCenter.CollectPerformanceData | Microsoft.SystemCenter.Library | Write performance data into the Operations Manager database. | System.Performance.Data |
| Microsoft.SystemCenter.DataWarehouse.PublishEventData | Microsoft.SystemCenter.DataWarehouse.Library | Write event data into the data warehouse. | System.Event.Data |
| Microsoft.SystemCenter.DataWarehouse.PublishPerformanceData | Microsoft.SystemCenter.DataWarehouse.Library | Write performance data into the data warehouse. | System.Performance.Data |
| Microsoft.Windows.PowerShellWriteAction | Microsoft.Windows.Library | Run a Windows PowerShell script. | System.BaseData |
| Microsoft.Windows.ScriptWriteAction | Microsoft.Windows.Library | Run a script. | System.BaseData |
| System.CommandExecuter | System.Library | Run a command line program | System.BaseData |
| System.Health.GenerateAlert | System.Health.Library | Generate an alert. | System.BaseData |

# Module Implementations

All modules are implemented in one of the following three ways:

 Native code

 Managed code

 Composite

As their names imply, managed code modules and native code modules represent managed code and native code installed on the agent computer. These modules are defined in management packs but must be installed separately on the agent computer. Modules that are defined in library management packs are installed automatically with the Operations Manager agent. The definition of a native module in the management pack just includes its Class ID. Managed modules include the details of their assembly. Creation or modification of native modules and managed modules is not supported, but those included in library management packs are available that can be used by modules and workflows in other management packs.

Composite modules are composed of one or more other modules. Composite modules are included in library management packs, and custom composite modules may be defined by management pack authors for performing custom logic. Composite modules require no installation on the agent computer and are completely defined in a particular management pack. They may include native code modules, managed code modules, other composite modules, or any combination of such modules.

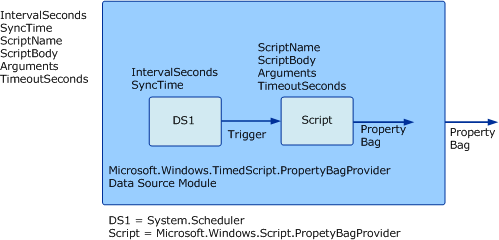
There is no difference between using modules with different implementations in a workflow or composite module. In fact, any workflow using a particular module is unaware how it is implemented or the details of how its functionality is performed. The workflow just has to know what function the module performs, its input data type requirement (if any), its output data type (if any), and its required parameters.

If you inspect the modules in a composite module and then continue to inspect any one of those modules that are composite, you will eventually drill down to a native or managed code module. These are the modules that perform core functionality on the agent whereas composite modules provide different variations on this functionality.

## Composite Module Example

To explain the concept of composite modules, the following diagram shows an example of a module included in the Microsoft Windows Library management pack. This is a common data source module named Microsoft.Windows.TimedScript.PropertyBagProvider. The function of this module is to run a script on a regular timed basis and return the results as a property bag. If you create a rule or monitor based on a script that uses a wizard in the Authoring console, this is the data source module that will be used in the resulting workflow.

Windows Timed Script Property Bag module



This module is a composite module that is actually composed of two other modules as follows:

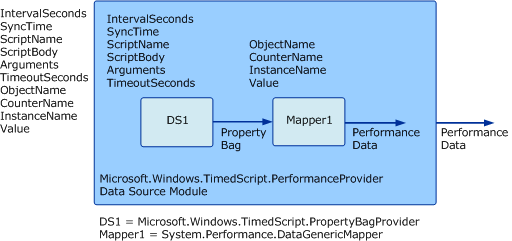
The System.SimpleScheduler module is a data source module that sends a trigger at set intervals. It requires parameters specifying the schedule that will be followed.

Microsoft.Windows.ScriptPropertyBagProbe is a probe action module that runs a specified script returning the output as a property bag. Because it is a probe action, it requires a trigger to start. That trigger is provided by the System.SimpleScheduler data source module. This module needs parameters for such information as the name of the script, the script body, and its arguments.

Both of the modules included in this composite module are composite modules themselves. However, this knowledge is not required in order to use them. The composite module that contains them just has to know what function the module will perform, what parameters it requires to perform that function, what input data type it requires, and what output data type it will generate. The internal details of how it performs this functionality are not relevant.

Another module in the same management pack library Microsoft.Windows.TimedScript.PerformanceProvider uses Microsoft.Windows.TimedScript.PropertyBagProvider but then maps its output from a property bag to performance data. This is performed by adding a condition detection module System.Performance.DataGenericMapper.

Windows Timed Script Performance module



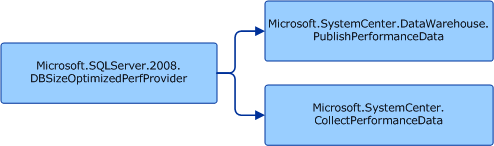
The advantage of this strategy is that the management pack author can select the module with the type of data output that is required. Instead of duplicate functionality in the management pack libraries, one module can just build on another. This same strategy can be leveraged in creating custom modules and workflows.

## Sample Workflow with Composite Modules

To additionally explain this concept of composite modules and how they can all be traced back to native or managed code modules, the following diagram shows a sample workflow from the SQL Server 2008 management pack. This is a rule named Microsoft.SQLServer.2008.Database.DBSpaceFree.Collection that runs a script to collect the free space for databases on a SQL Server 2008 instance.

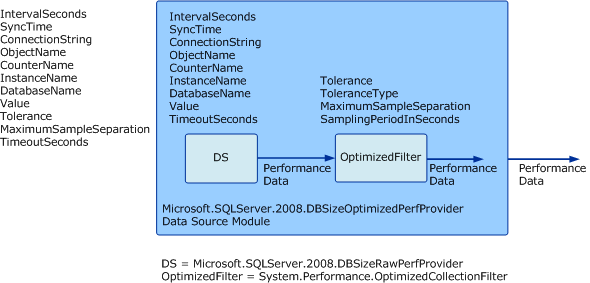
The workflow has a single data source module named Microsoft.SQLServer.2008.DBSizeOptimizedPerfProvider that runs the required script, formats it into performance data, and optimizes it for efficient collection. Then it stores the collected data into the Operations Manager database and the data warehouse using two write action modules.

SQL DB Size Optimized Performance rule



Each of the write action modules are native modules. Neither requires any input parameters but can just work with performance data output from the previous module. The data source module is a custom composite module in the same management pack. That module is shown in the following diagram. It uses another data source module named Microsoft.SQLServer.2008.DBSizeRawPerfProvider that collects the free space data and outputs it as performance data. This data is sent to a native condition detection module System.Performance.OptimizedCollectionFilter that optimizes the collected performance data. Its function is to determine whether the collected value has deviated sufficiently from the previously collected value in order to pass it along for collection.

SQL Server Database Size Optimized module



Microsoft.SQLServer.2008.DBSizeRawPerfProvider is another composite module made up of three modules. The first is a data source module that runs the script on a regular schedule and sends output as a property bag. This is followed by a condition detection module that maps the property bag data to performance data. A second condition detection module filters the property bag according to the database name. This is required because the script returns multiple property bags, one for each database on the SQL instance. This strategy is used to support [Cookdown](#z1f8fb91b1e814c8daacf758edb2f3141), which is discussed in a separate section. The two condition detection modules are native modules, and the data source module is the same composite module discussed previously.

# Kinds of Workflows

Each kind of workflow in an Operations Manager 2007 management pack must follow specific criteria according to the modules types they can contain. The following sections provide details on each kind of workflow.

## Discoveries

Discoveries are a special type of rule that are used for the sole purpose of collecting discovery data to be inserted into the Operations Manager database.

A discovery has only a single data source module that must output discovery data. If more complex logic is required that uses other modules, such as a condition detection, a composite data source module must be created and used by the discovery.

Discovery workflow



## Rules

Rules are typically used for one of the following purposes:

 Collect and store data in either the Operations Manager database, the data warehouse, or both.

 Generate an alert based on collected data.

 Run a script or command according to a regular schedule.

A rule is defined by the following:

 One or more data source modules.

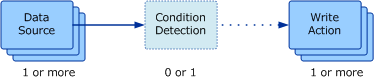
 Zero or one condition detection modules.

A condition detection module is required if more than one data source module is used. In this case, the condition detection module would collect data from the different data source modules in order to produce a single output.

 One or more write action modules.

Each write action module accepts the same data output from the data source module or the condition detection module if present.

Rule workflow

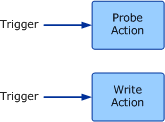


## Tasks

Tasks are workflows that are not loaded by the agent until they are run by an operator.

A task is defined by using a single module. If the task is only collecting information, a probe action module should be used. If the task is modifying the system, a write action module should be used.

Task workflow



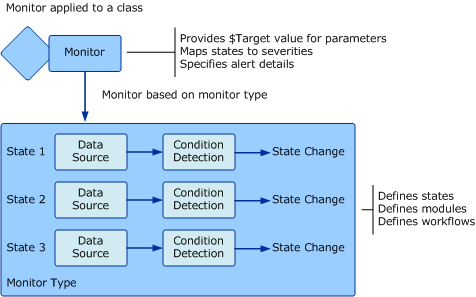
## Monitors

Monitors are workflows used to determine the state of monitored objects. Each monitor has either two or three potential states and is in a single state at any particular time. Aggregate monitors and dependency monitors derive their states from one or more unit monitors. They are workflows, but their details are not exposed to the management pack author. Unit monitors are the only type of monitor that the management pack author will compose.

### Monitor Types

Monitors are based on monitor types. Monitor types define the modules and logic that are used in the workflow. They have parameters like modules so that a single monitor type can be shared by multiple modules with each monitor providing values specific to the particular scenario. Each monitor represents the actual workflow that targets a particular class and provides values to the monitor type parameters.

Monitor and monitor type



A single monitor type can detect state using both regular detection and the optional on-demand detection. Regular detection is the normal operation of the monitor as it continuously polls for information to determine the state of the monitored object. On-demand detection occurs when the monitor is first initialized on an agent, when the monitored object returns from maintenance mode, and when the operator selects Reset Health for the monitor in the Operations console. If on-demand detection is not defined for a monitor type, the state is not immediately set from these events but is defined the next time that regular detection occurs.

Each health state of the monitor has its own workflow. For example, if a particular monitor type has three states and defines both regular detection and on-demand detection, that monitor type will have six different workflows. While the end result of each workflow is to set the state of the same monitor, each workflow operates completely independently of the others. Each one is working to determine whether it should set the state of the monitor to the workflow’s specific state. Because of this, the management pack author must ensure that workflows for different states can never resolve to true at the same time. If this situation does occur, the results on the health state of the monitor are unpredictable.

The workflow for each health state for regular detection for a monitor type is defined by the following:

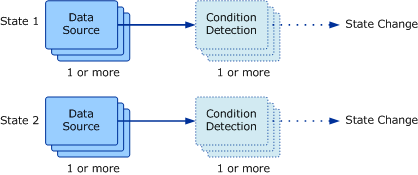
 Must begin with one or more data source modules.

 May include probe action modules and condition detection modules in any order.

 Only a single data stream can be output. A condition detection module is required if more than one data source module or probe action module is used. In this case, the condition detection module would collect data from the different modules in order to produce a single output.

 May not include a write action module. The only result of a workflow in a monitor type is to set the health state of the monitored object, so there is no purpose for a write action module.

Monitor type regular detection



The workflow for each health state for regular detection for a monitor type is defined by the following:

 Must begin with a probe action module that requires no input data.

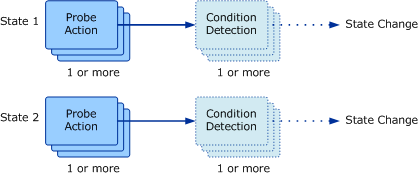
 May include probe action modules and condition detection modules in any order.

 Only a single data stream can be output. A condition detection module is required if more than one data source module or probe action module is used. In this case, the condition detection module would collect data from the different modules in order to produce a single output.

 May not include a data source module. Data source modules are not able to operate on demand.

 May not include a write action module. The only result of a workflow in a monitor type is to set the health state of the monitored object, so there is no purpose for a write action module.

Monitor type on demand detection



### Diagnostics

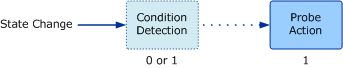
Diagnostics are special kinds of tasks that collect information about a particular monitor. Diagnostics can be run automatically when the monitor changes state, or manually by an operator when the monitor is in a given state. It is not intended to make any changes to the system.

A diagnostic is defined by the following:

 Zero or one condition detection module.

 One probe action module.

Diagnostic workflow



### Recoveries

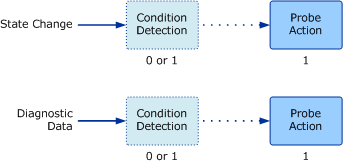
Recoveries, just as diagnostics, are special tasks that are triggered by a state change in a monitor. Recoveries can be run automatically when the monitor changes state, or manually by an operator when the monitor is in a given state. Whereas a diagnostic is intended to collect information, the function of a recovery is to try to resolve the problem detected by the particular monitor. Or, a recovery may run only after a particular diagnostic runs and returns data matching specific criteria.

A recovery is defined by the following:

 Zero or one condition detection module.

 One write action module.

Recovery workflow



# Cookdown

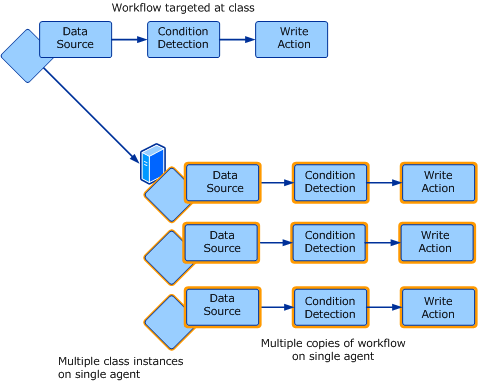
Cookdown refers to a feature of Operations Manager 2007 where a single copy of a data source module is shared among multiple workflows to reduce overhead. Understanding how cookdown works and how to design workflows to take advantage of cookdown can be important to making sure that a particular workflow operates in the most efficient manner and does not place too much overhead on the managed agent.

## Overview of Cookdown

### Multiple Workflows

The Operations Manager agent will load a separate copy of a workflow for each instance of the target class. Most agents manage a considerable number of objects each with several workflows targeted at them. This means that several workflows are typically running on any given agent. The Operations Manager agent can run multiple workflows efficiently, so that this is typically not a concern. However, Data Source modules often run processes outside of the Operations Manager agent that have the potential for considerable overhead. The most obvious example of this is a script that can potentially generate significant overhead, depending on what actions it is performing. If a workflow running that script is targeted at a class that has multiple instances on an agent, that agent will be running multiple instances of the script at the same time. As the number of instances of the class increases, the number of instances of the script increases. These could generate significant overhead on the agent.

Multiple instances of workflow on a single agent

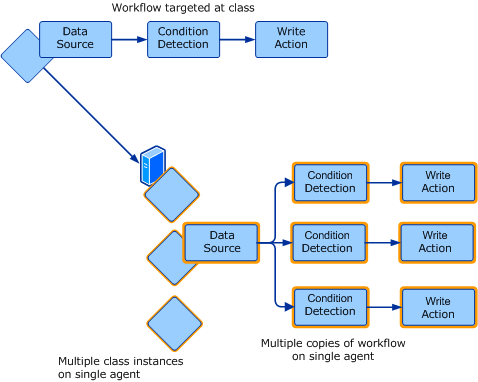


For example, the Windows Server 2008 management pack has a monitor called Microsoft.Windows.Server.2008.LogicalDisk.FreeSpace (display name Logical Disk Free Space) that runs a script to measure the free space on each logical disk on a managed agent. This monitor is targeted at the Windows Server 2008 Logical Disk class, and will have multiple instances on any agent that has more than one logical disk defined. The monitor has three states, meaning that the agent will load a copy of three workflows for each instance of the target class. This means that an agent with four logical disks would have 12 different workflows. Without cookdown, every time the monitor was scheduled to run, 12 copies of the script would run at the same time.

### Multiple Workflows with Cookdown

With cookdown, the agent still runs a separate workflow for each instance of the target class. However, it loads only a single instance of the data source sharing the output with the different workflows. This is a significant reduction in potential overhead.

Multiple instances of workflow sharing a single data source

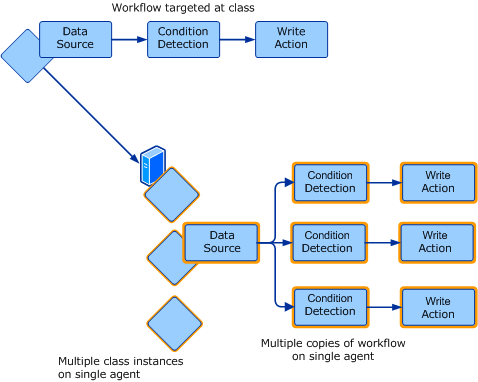


Only data source modules are cooked down. However, composite data source modules may contain modules of other types such as probe action modules and condition detection modules. If such a composite data source module supports cookdown, it will be cooked down. Because only a single instance of the data source module is loaded, only a single instance of the modules within it is also loaded.

The previously mentioned monitor Microsoft.Windows.Server.2008.LogicalDisk.FreeSpace supports cookdown, and only a single instance of its data source is loaded regardless of the number of logical disks on the agent. A separate set of workflows are still loaded for each agent, yet they all share the single copy of the data source module. Because the data source module runs the script, the result is a significant reduction in the overhead generated by the different instances of the monitor.

In addition to multiple copies of the same workflow, cookdown applies to different workflows sharing the data source module. For example, one data source module running a script might be used by a monitor for measuring health state and also used by a rule for collecting performance data. Cookdown could be performed in such a way that a single copy of the data source module would be shared between the different workflows.

Multiple instances of workflow sharing single data source module



## Criteria for Cookdown

A workflow does not have to specify whether cookdown should be performed. Cookdown is performed automatically on any workflow that meets the cookdown criteria: the only criterion is that all copies of the data source module are called by using identical values for each parameter. If this is the case, all instances of the workflow on each agent will cookdown to a single data source module.

The only way that values for a parameter will vary for different instances of the same module is if a $Target variable is used. This kind of variable uses the value of a property on the target object. Because this value may vary between different instances, the value that is provided to the module parameter could vary. In this case, cookdown would not be performed, and a separate copy of the data source would be used for each workflow.

Values that are provided to the parameters for a data source module shared between different kinds of workflows have more potential to vary. For example, a rule and a monitor might share a data source module running a script. The interval that the script should run is provided to the data source module as a parameter. If the monitor and rule are configured to run at different intervals, the value that is provided to that parameter will vary between the workflows. In this case, cookdown would not be performed, and a separate copy of the data source module would be required for the different workflows.

## When to Configure for Cookdown

Changing a script and a workflow to support cookdown can take significant effort, and that effort is not valuable in all circumstances. You should be concerned about cookdown only when multiple instances of the target class are expected on a single agent. When there is only a single instance of the target class expected on the agent, cookdown will not be performed anyway because there will only be a single copy of the workflow.

# Scripts Supporting Cookdown

Frequently, a script in a data source module will require the value of a property from the target object. The most straightforward method of providing this value would be to use a $Target variable in a parameter to the module, but then cookdown cannot be performed. The method for supporting this type of scenario is to have the script collect information for all instances of the target class and return a property bag for each. Because the script itself becomes responsible for identifying the values of the individual properties, you do not have to provide them through parameters to the module. Because there are no $Target variables in the parameters, cookdown is preserved.

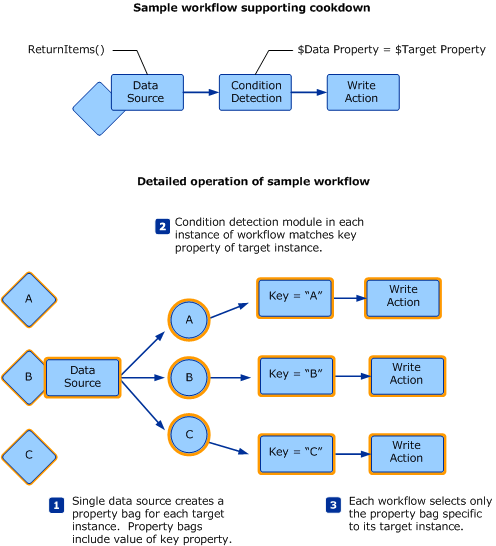
## Multiple Property Bags

For the script to perform cookdown, it will frequently re-create some basic logic used in a discovery script for the target class. The discovery script is typically responsible for identifying the different instances and collecting values for the different properties. Whereas adding this logic makes the script itself more complex and requires more overhead than a script written for only a single instance, the script has the advantage of supporting cookdown. One copy of the script running for all instances of the target class typically generates significantly less overhead than multiple copies of a script running for each single instance.

A script can return multiple property bags by using the ReturnItems method on the MOM.ScriptAPI object instead of Return. This returns a single property bag. When a script returns multiple property bags, each one is collected and processed by the next module in the workflow. If a method of filtering is not used, each workflow will incorrectly complete for each instance. This can result in unwanted situations such as all monitors going to an error state when an error condition is detected on a single instance or collecting multiple duplicate performance values for each instance.

To have a workflow filter for the property bag matching its target instance, a condition detection module is used after the data source module or probe action module running the script. This condition detection module uses a $Target variable for the key property of the target instance. The property bag itself must include the same value in one of its properties. The condition detection is then able to match the value from the property bag to the value of the $Target property to let each copy of the workflow to select the property bag specific to its particular instance. This concept is illustrated in the following diagram.

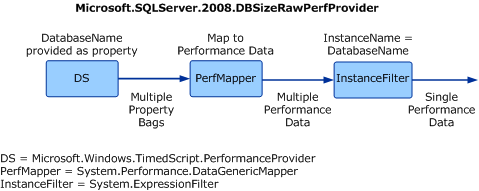
Workflow supporting cookdown



The module Microsoft.SQLServer.2008.DBSizeRawPerfProvider described in [Module Implementations](#zb3f7fae7291c49a9b153ef4cc51370aa) provides an example of a script creating multiple property bags to support cookdown. The DatabaseName of the target instance is sent to the module; the name is not passed to the Microsoft.Windows.TimedScript.PerformanceProvider module which runs the actual script. This module can cookdown to a single instance, although the modulecreates a separate property bag for each database. Each property bag includes a value for the name of the database. The value of the DatabaseName parameter is sent to the System.Expression module that compares it to the value in the property bag. Using this method, all instances of a workflow on a particular agent that uses this module will be able to share a single copy of the data source module yet will be able to select only the property bag specific to the particular instance.

The following is a simplified view of the Microsoft.SQLServer.2008.DBSizeRawPerfProvider module showing this concept.

SQL Database Raw Performance module using cookdown



# Designing Custom Modules and Workflows

Many workflows, such as rules, monitors, and discoveries, can be created using wizards in the Authoring console. When these wizards are used, minimal knowledge of the underlying modules and data stream is required. The Authoring console wizard helps you construct a workflow addressing a specific scenario with minimal input from the user. The wizard does this by selecting from a specific set of modules and by using user-specified information for parameters. Once the workflow is created though, the individual modules can be viewed and changed as if it was created manually.

The Authoring console does provide functionality for manually creating workflows, although some modules may require direct editing of the XML by using an editor. When a custom workflow is created, you have to select the individual modules that will be used and provide values or variables for their parameters. Depending on the kind of workflow you are creating, there may be other considerations also, as identified in the following sections.

## When to Create a Custom Workflow

### Functionality Not Provided with Wizard

The wizards in the Authoring console create workflows based on a specific set of scenarios. If you must have logic outside these scenarios, you must create a custom workflow. The workflow may use custom modules or the workflow may be composed completely of modules from management pack libraries.

For example, the Authoring console provides a number of complex scenarios for wizards that use events. These include correlating multiple events, detecting missing events, and waiting for multiple occurrences of the same event. These options are not provided for rules however, and those wizards are limited to more basic scenarios. A custom rule would have to be created to use any of this logic. The rule could be created from the identical set of modules used by the monitor types that the wizards are based on.

### Custom Modules

Wizards in the Authoring console use specific modules that are defined in management pack libraries. If a workflow requires a custom module, you have to create a custom workflow. Those reasons why you might create a custom module are detailed in [Custom Module Types](#zc7d836343ae5494fb5fb8708465578f7).

A common scenario for a custom workflow is to share a script between a monitor and a rule with the monitor measuring health state and the rule collecting data for reporting and analysis. This scenario is achieved by creating a custom data source module or probe module to contain the script and then creating a custom rule, custom monitor type, and custom monitor to use the new module.

### Windows PowerShell

Operations Manager 2007 R2 includes modules for running Windows PowerShell scripts. There are no wizards in the Authoring console that will take advantage of these modules, so they may only be accessed by creating a custom workflow.

## When to Create Custom Modules

If an existing module will achieve the requirements for a particular workflow, it can typically be used without creating a custom module. A custom composite module may be useful to address the conditions described here.

### Share Common Logic between Different Workflows

A single module can be used by multiple workflows. If a script or some other complex logic has to be shared by multiple workflows, a custom module can be created to encapsulate the logic. Instead of implementing the logic separately in each workflow, the custom module can be shared by each.

For example, consider a script that is used to collect performance data for a particular application. A monitor could be used to set the health state of a monitored object according to the collected data’s comparison with a particular threshold. In addition, a rule could be used to collect the performance data for analysis and reporting. Without a custom module, the script would have to be individually provided as a parameter to an existing module in both the rule and the monitor. If that script were included in a custom module however, it might be shared by both the monitor and the rule. The script would only have to be created and maintained in one location in the management pack.

### Implement Complex Logic Not Possible with Existing Modules

Custom workflows can be created to implement complex logic with existing modules. However, different kinds of workflows have specific requirements as to the types and number of modules that they contain. If the logic cannot be created within these restrictions, custom modules can be created to encapsulate this logic and make it valid for the particular workflow.

For example, a particular discovery might require a data source module in addition to a condition detection module. However, a discovery is allowed only a single data source module. A custom module could be created that contains the required data source and condition detection modules. The discovery would use the custom data source module without exposure to the modules that it contained.

### Improve Override Experience

Custom modules can define parameters that accept values from the workflow. These are exposed to the end-user of the management pack through overrides where they can be changed. If a particular module has a complex parameter that may not be intuitive to the user, the module may be wrapped in a composite module that provides more intuitive parameters. The values are then passed on to the required parameters through $Config variables.

For example, the modules to run a script will typically have a parameter called Arguments that contains a space- delimited list of arguments to be provided to the script. If the user needed to override one or more of these values, they would have to be exposed to the complete Arguments parameter and determine which particular part of it needed to be modified. For a script that uses multiple arguments, the required formatting will most likely not be intuitive to the end-user. An alternative strategy would be to create a custom module with parameters by using descriptive names. This module would contain the module running the script that would build the Argument parameter through the appropriate combination of $Config variables.

# Custom Module Types

Custom module types are not as restrictive as custom workflows. Each type has requirements as to the kinds of modules they can contain, and whether they require input or output data. Within those guidelines though, the custom module can contain multiple modules arranged in any specified order.

## Data Source Modules

Data source modules are the most common type of custom module, because they are used for rules, regular detection on monitors, and discoveries. They frequently contain custom scripts and other logic to retrieve data from a particular application or device.

Data source modules can contain modules of the following types:

 Data source modules

 Probe action modules

 Condition detection modules

Data source modules have the following requirements:

 Must start with at least one data source module.

 Must deliver output in a single data stream. If the module contains multiple data source modules or multiple probe action modules, one or more condition detection modules must be used to consolidate or filter the data streams.

 The type of the output data will match the type output by the last included module.

## Probe Action Modules

Probe action modules are used in tasks and in monitors for on-demand detection. Like data source modules, they can use to custom scripts and other logic to retrieve data from a particular application or device. The difference is that a probe action cannot be used without being embedded in a data source module in rules and in monitor regular detection. A common strategy is to use a probe action module to hold the script and then encapsulate that module in a data source module that adds a scheduler. This strategy allows the same logic to be used for rules, both on demand and regular detection monitors, and tasks.

Probe action modules can contain modules of the following types:

 Probe action modules

 Condition detection modules

Probe action modules have the following requirements:

 Must declare an input data type that matches the required input of the first module.

 Must deliver output in a single data stream. If the module contains multiple probe action modules, one or more condition detection modules must be used to consolidate or filter the data streams.

 The type of the data will match the type output by the last included module.

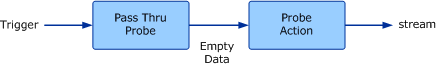
### Probe Action Module Input Data and Triggers

Probe action modules will either require input data from a previous module or require only a trigger without data provided. Only those that are configured as Trigger Only may be used in on-demand detection for a monitor type. If a composite probe action module contains another probe action module that requires input data, however, that module must also require input data.

This may present an issue if a probe action module is required for a monitor type that uses on-demand detection but a Trigger Only probe action module is not available. The most obvious example of this is the probe action module for running a script Microsoft.Windows.ScriptPropertyBagProbe that does require input.

The solution to this issue is a special probe action module named System.PassThroughProbe. This module requires no input, performs no function, and outputs blank data. It may be used at the start of any composite probe action module that has to be Trigger Only but must also include another module requiring input.

Pass through probe



## Condition Detection Modules

Custom condition detection modules are not common, although they can be created for specific scenarios.

Condition detection modules can contain modules of the following types:

 Condition detection modules

Condition detection modules have the following requirements:

 Must declare one or more input types that match the type of the expected data.

 Must deliver output in a single data stream.

 The type of the data will match the type output by the last included module.

## Write Action Modules

Write action modules are used in rules to perform some final action of a workflow or in tasks that perform some action at the request of the user. The data from a write action module is never sent into the data stream. If data is output from a write action module, it is stored in the Operations Manager database or data warehouse. Custom write action modules are frequently created to contain scripts that perform some action against a particular application.

Write action modules can contain modules of the following types:

 Write action modules

Write action modules have the following requirements:

 Must declare one or more input types that match the type of the expected data.

 Must include at least one write action module.

# Passing data with parameters

Most modules and monitor types will have parameters in order to accept information from the defined workflow and from other modules in the data stream. Workflows and composite modules must provide values for the required parameters of the modules that they contain and may provide values for optional parameters.

## Values and Variables

Values that are provided to parameters will be a combination of explicit values, $Config variables, $Target variables, and $Data variables. Each of these is discussed in the following sections with recommendations on when each should be used.

### Explicit Values

Explicit values for parameters are just values that do not change regardless of the target object or incoming data. Examples include the threshold value for a monitor, an event number that indicate a particular error, or the interval that a script should run.

Explicit values are typically provided by workflows to monitor types and modules. For example, a module running a timed script will require a parameter defining how frequently it should run. A rule that uses this module would have to provide an explicit value such as 900 seconds. This means that the module would run every 15 minutes. In this context, a variable wouldn’t make any sense, and all copies of the workflow would be run on the same 15-minute schedule.

The definition of modules and monitor types will also use explicit values for parameters where a particular value will not change. For example, a composite module might be created in order to share a particular script between rules and monitors. One of the modules that is contained by this composite module would be responsible for running the script. Values for parameters that are required by this module, such as the name of the script and the body of the script, would not vary between different target objects and would be explicitly defined in the definition of the module.

### $Target Variables

$Target variables refer to the value of a property on the target object or one of the target object’s hosting parents. These values are collected by the discovery process for the target class and will not be known when the workflow is defined. The workflow will also run against all instances of the target object, and each may have a different value for the particular property.

$Target variables are most frequently used by workflows sending values to monitor types and modules that accept the value that uses a $Config variable. For example, a rule collecting a performance counter might be targeted at a class where multiple instances are expected on each agent. To distinguish each collected value, the instance name of the performance data could be populated with the name or some other key property of each object. This value would be provided through a $Target variable that would provide a unique value for each target object.

### $Data Variables

$Data variables refer to some value from the data incoming from the previous module. The format of the variable will depend on the type of data being used. Example uses of $Data variables are in expressions that inspect the incoming data to determine whether some action should be taken, in the mapping of data from one type to another, and in write action modules generating alerts where incoming data values are used in the alert description.

For example, a common scenario is a monitor that runs a script and uses the resulting data to determine the health state of the target object. The workflow for each state (of the possible health states) would use a condition detection module with an expression comparing the resulting data to different specified values. These expressions would use a $Data variable to access the required value.

### $Config Variables

$Config variables are used to reference the value that is provided to the module or monitor type that contains the particular module. Values for parameters of a module inside a composite module will typically be either explicit values or $Config variables. This is a means of having the workflow provide the value for a parameter that is then passed to a module inside the composite module.

Although parameters that are passed through to contained modules typically map directly between parameters, this is not a requirement. $Config parameters can actually be used to hide the complexity of underlying parameters from the workflow. This can be valuable in providing users with more straightforward parameters exposed to overrides. For example, consider a composite module that contains the data source module Microsoft.TimedScriptPropertyBagProvider. That module has a parameter called Arguments that requires the arguments line that is provided to the script when it is run. The script may require several parameters, not all of which should be overridden by the end-user. Instead of defining a parameter called Arguments on the composite module, a separate parameter could be provided for individual arguments. The Arguments parameter on the module would be composed of multiple $Config variables.

## Overrideable Parameters

Users of a management pack can change the values being passed to modules in the workflow by using overrides. Only parameters that are specified as overrideable are available to be overridden. The user may not change the value of any other variables. The definition of parameters that can be overridden is not made on the workflow itself but is made on the module or monitor type that the workflow is based on. In the Authoring console, parameters that can be overridden are specified on the Overrideable Parameters tab of the module properties.

Some parameters are always available to be overridden and do not have to be specified. These are available to the user as parameters that can be overridden but are not specified as parameters on the module or monitor type.

The following table shows workflows that have the following parameters that can be overridden.

|  |  |
| --- | --- |
| Enabled | Enable or disable the workflow. |

Any monitor or rule that creates an alert has several parameters specific to the alert that can be overridden. The following parameters on rules that generate alerts can be overridden:

|  |  |
| --- | --- |
| Priority | Priority of the alert. |
| Severity | Severity of the alert. |

As shown in the following table, monitors that generate alerts will have these parameters:

|  |  |
| --- | --- |
| Alert On State | Monitor state that generates the alert. |
| Alert Priority | Priority of the alert. |
| Alert severity | Severity of the alert. |
| Auto-Resolve State | Monitor state that automatically resolves the alert. |
| Generates Alert | Specifies whether an alert should be generated in response to a state change for the monitor. |

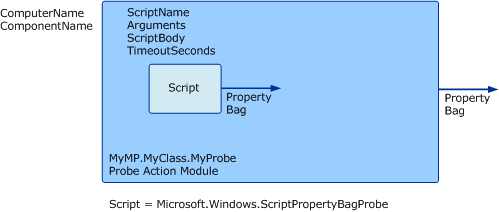
## Sample Workflow

The following sample workflow shows the different values and variables provided to the parameters of different modules. This represents a rule based on a script collecting performance data. The script requires two arguments for the name of the computer and the name of the component that the module is running against. The script itself is included in a probe action module. This module is used in a data source module that adds a scheduler in order to run the script on a timed basis. This data source module is then included in another data source module that maps the property bag from the script to performance data.

Although this may seem to be a complex strategy that uses three different modules for functionality that might be completed by using a wizard in the Authoring console, it is actually a common strategy to support multiple scenarios sharing the same script. The probe action module could be used in a task or in on-demand detection for a monitor. The two data source modules could be used in a rule or regular detection for a monitor. Each provides a different kind of output that depends on the requirements for the particular workflow.

The custom probe action module uses Microsoft.Windows.ScriptPropertyBagProbe (alias Script) to run the actual script, as is shown here.

Custom probe action module

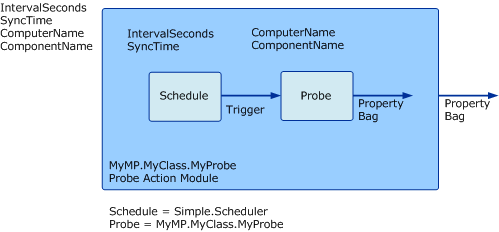


The following table lists the values that are provided for the parameters to the module:

|  |  |  |  |
| --- | --- | --- | --- |
| Module | Parameter Name | Value Used | Details |
| Script | ScriptName | MyScript.vbs | Explicit value, because the name of the script will not vary for different copies of the workflow. |
| Arguments | $Config/ComputerName$ "$Config/ComponentName$" | These are the arguments that are provided on the command line of the script. These are variables because they will vary, depending on the agent and target object the workflow is running against. $Config variables are used so that different workflows can provide the required values. Quotation marks are used around the ComponentName variable in case its value contains spaces. |
| ScriptBody | <Body of the script> | Explicit value, because the body of the script will not vary for different copies of the workflow. |
| TimeoutSeconds | 300 | Explicit value, because the timeout for the script will not vary for different copies of the workflow. |

The first data source module uses System.Scheduler (alias Schedule) followed by the custom probe module (alias Probe). This is shown here:

Custom data source module outputting property bag

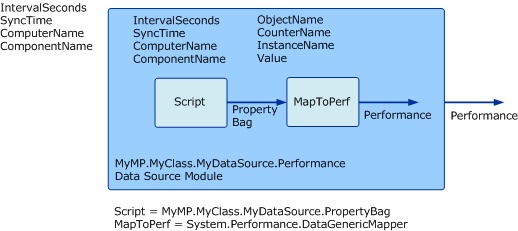


The values that are provided for the parameters to the modules are shown on the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Module | Parameter Name | Value Used | Details |
| Schedule | IntervalSeconds | $Config/IntervalSeconds$ | $Config variable used so that the workflow can provide its own schedule. |
|  | SyncTime | $Config/SyncTime$ | $Config variable used so that the workflow can provide its own schedule. |
| Probe | ComputerName | $Config/ComputerName$ | $Config variable used so that the workflow can provide the computer name. |
|  | ComponentName | $Config/ComponentName$ | $Config variable used so that the workflow can provide the component name. |

The second data source module uses the first data source module (alias Script) followed by the condition detection module System.Performance.DataGenericMapper (alias MapToPerf), as illustrated here.

Custom data source module outputting property bag

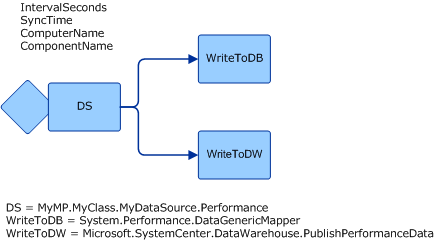


The following table shows the values that are provided for the parameters to the modules:

|  |  |  |  |
| --- | --- | --- | --- |
| Module | Parameter Name | Value Used | Details |
| Script | IntervalSeconds | $Config/IntervalSeconds$ | $Config variable used so that the workflow can provide its own schedule. |
| SyncTime | $Config/SyncTime$ | $Config variable used so that the workflow can provide its own schedule. |
| MapToPerf | ObjectName | MyApp | Explicit value, because the name of the object for the performance data will not vary for different copies of the workflow. |
| CounterName | MyCounter | Explicit value, because the name of the counter for the performance data will not vary for different copies of the workflow. |
| InstanceName | $Data/Property[@Name='ComponentName']$ | $Data variable that refers to a value from the property bag generated by the previous module. The script would be required to create the property bag with a value with the name ComponentName. |
| InstanceName | $Data/Property[@Name='Value']$ | $Data variable that refers to a value from the property bag generated by the previous module. The script would be required to create the property bag with a value with the name Value. |

The rule uses the second data source module (alias DS) followed by two write action modules to store the data in the Operations Manager database and the data warehouse. There is no requirement for a condition detection module in the rule because the required modules for filtering and mapping to a different data type were included with composite modules. The write action modules do not require any parameters. This is shown here:

Custom rule



The values that are provided for the parameters to the module are shown in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Module | Parameter Name | Value Used | Details |
| DS | IntervalSeconds | 900 | Run the script every 15 minutes. |
| SyncTime |  | Left blank, because no synchronization time is required. |
| ComputerName | $Target/Host/Host/Property[Type= "Windows!Microsoft.Windows.Computer"]/PrincipalName$ | $Target variable to use the computer name of the agent hosting the target object. This variable assumes that the target class is hosted by a class that is hosted by Windows Computer. |
| ComponentName | $Target/Property[Type="MyMP.MyApplicationComponent"]/ComponentName$ | $Target variable to use the component name of the target object. This variable assumes that the target class has a property named ComponentName. |

# Creating Custom Modules and Workflows

The following procedures provide guidance on how to create custom modules and workflows in the Operations Manager 2007 Authoring console.

## In This Section

[How to create a monitor based on a custom module](#zf0eaf72536184c46bc19db226315f4cc)

|  |
| --- |
| Create a monitor and monitor type based on a custom data source module and a custom probe action module that contains a script. |

[How to create a monitor based on a Windows PowerShell script](#z64c66f4abce24bb793c449de3be6035b)

|  |
| --- |
| Create a monitor and monitor type based on a custom data source module and a custom probe action module that contains a Windows PowerShell script. |

[How to create a monitor and rule that share a script supporting cookdown](#z985beef895384ca2bd14aea5275426ea)

|  |
| --- |
| Create a monitor and monitor type based on custom data source module that contains a script supporting cookdown. |

# How to create a monitor based on a custom module

The following procedure shows how to create a monitor based on a custom data source module running a script using Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the class to act as the target.

The monitor has the following characteristics:

 Targeted at a class with only a single instance on an agent. Because there is a single instance, there is no requirement to support cookdown.

 Sets its state based on the comparison of the script value to specified threshold values.

 The monitor supports On Demand Discovery. This requires a probe action module to run the script.

 The script accepts only a single argument for the computer name of the target object’s agent.

 The script itself is only for testing and performs no real function. It simulates a script running a synthetic transaction and returning a status message of success or failure.

To create probe action module to run a script

|  |
| --- |
| 1. Select Type Library and then Probe Actions.  2. Right-click in the Probe Actions pane and select New and then Composite Probe Action.  3. In the Choose a unique identifier box, type MyMP.ProbeActionModule.MyTransactionScript. Click OK.  4. On the General tab, in the Name box, type Transaction Script Probe Action.  5. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select Microsoft.Windows.ScriptPropertyBagProbe.  c. In the Module ID box, type Script. Click OK.  d. In the ScriptName box, type MyTransactionScript.vbs.  e. In the Arguments box, type $Config/ComputerName$.  f. In the TimeoutSeconds box, type 300.  g. Click the Edit… button. This starts the custom editor.  h. Paste the complete contents of the following script between the ScriptBody tags in the XML. Replace any text that might already exist.  <![CDATA[  sComputerName = WScript.Arguments(0)  bTestSuccessful = True  Set oAPI = CreateObject("MOM.ScriptAPI")  oAPI.LogScriptEvent "MyTransactionScript.vbs",10,4, "Running script on " & sComputerName  Set oBag = oAPI.CreatePropertyBag()  Call oBag.AddValue("ComputerName",sComputerName)  If bTestSuccessful = True Then  Call oBag.AddValue("Result","Good")  Else  Call oBag.AddValue("Result","Bad")  End If  oAPI.Return(oBag)  ]]>  i. Close the editor to save the script back to the module.  j. Click OK to save the module configuration.  k. In the NextModule column for the Script module, select Module Output.  6. On the Configuration Schema tab, do the following:  a. In the Simple Configuration Schema section, click Add to add a parameter.  b. In the Please enter the requested value box, type ComputerName. Click OK.  7. On the Data Types tab, do the following:  a. In the Input Data section, select This module required input data.  b. In the Output Data section, in the Data Type: box select System.PropertyBagData.  8. Click OK to save the module.  9. Select File, and then click Save. |

To create data source module to run probe action on schedule

|  |
| --- |
| 1. Select Type Library and then Data Sources.  2. Right-click the Data Sources pane, select New, and then Composite Data Source.  3. In the Choose a unique identifier box, type MyMP.DataSourceModule.MyTransactionScriptTimed. Click OK.  4. On the General tab, in the Name box, type Timed Transaction Script Data Source  5. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select System.SimpleScheduler.  c. In the Module ID box, type Schedule. Click OK.  d. Click the button to the right side of the IntervalSeconds box and select Promote. This enters the text $Config/IntervalSeconds$  e. Click the button to the right side of the SyncTime box and select Promote. This enters the text $Config/SyncTime$  f. Click OK to save the module configuration.  g. Click Add to add a new module.  h. In the Choose Module Type box, select MyMP.ProbeActionModule.MyTransactionScript.  i. In the Module ID box, type Probe. Click OK.  j. Select the button to the right side of the ComputerName box and select Promote. This enters the text $Config/ComputerName$.  k. Click OK to save the module configuration.  l. In the NextModule column for the Schedule module, select Probe.  m. In the NextModule column for the Probe module, select Module Output.  6. On the Configuration Schema tab, do the following:  a. Change the Type for the IntervalSeconds parameter to Integer.  b. Uncheck the Required box next to the SyncTime parameter. The SyncTime parameter is optional for this module.  7. On the Overrideable Parameters tab, do the following:  a. Click Add and then IntervalSeconds.  b. In the Choose a unique identifier box, type IntervalSeconds. Click OK.  c. Change Configuration Element to Integer.  d. Click Add and then SyncTime.  e. In the Choose a unique identifier box, type SyncTime. Click OK.  8. On the Data Types tab ensure the value in the Data Types box is System.PropertyBagData.  9. Click OK to save the module.  10. Select File, and then click Save. |

To create monitor type using custom data source

|  |
| --- |
| 1. Select Type Library and then Monitor Types.  2. Right-click in the Monitor Types pane and select New and then Composite Monitor Type.  3. In the Choose a unique identifier box, type MyMP.MyTransactionMonitorType. Click OK.  4. On the General tab, in the Name box, type My Transaction Script Monitor Type.  5. On the States tab, do the following:  a. Select 2 State Monitor Type.  b. In the ID of state 1 box type Success.  c. In the ID of state 2 box, type Failure.  6. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select System.PassThroughProbe.  c. In the Module ID box, type PassThru. Click OK.  d. Click OK to save the module configuration.  e. Click Add to add a module.  f. In the Choose Module Type box, select MyMP.ProbeActionModule.MyTransactionScript.  g. In the Module ID box, type Probe. Click OK.  h. Click the button to the right of the ComputerName box and select Promote. This will enter the text $Config/ComputerName$  i. Click OK to save the module configuration.  j. Click Add to add a new module.  k. In the Choose Module Type box, select MyMP.DataSourceModule.MyTransactionScriptTimed.  l. In the Module ID box, type DataSource. Click OK.  m. Click the button to the right side of the IntervalSeconds box and select Promote . This enters the text $Config/IntervalSeconds$  n. Click the button to the right side of the ComputerName box and select Promote. This enters the text $Config/ComputerName$  o.  Click Edit. This starts the custom editor.  p. After the line  <IntervalSeconds>$Config/IntervalSeconds$</IntervalSeconds>  Add the following line  <SyncTime>$Config/SyncTime$</SyncTime>  q. Close the editor to add the XML back the Authoring console.  Note  If you receive an error saying that the IntervalSeconds parameter is invalid according to its data type, click Ignore. This error message occurs because the IntervalSeconds parameter is configured as an integer, and the Authoring Console is reading $Config/IntervalSeconds$ as a string. This variable will be replaced with an integer value when the workflow is run so the error can be ignored.  r. Click OK to save the module configuration.  s. Click Add to add a new module.  t. In the Choose Module Type box, select System.ExpressionFilter.  u. In the Module ID box, type FilterSuccess. Click OK.  v. Click Configure to open the Expression dialog box.  w. Click Insert.  x. In the Parameter Name box type Property[@Name='Result'].  y. In the Operator box select Equals.  z. In the Value box type Good.  aa. Click OK to save the expression.  bb. Click OK to save the module configuration.  cc. Click Add to add a new module.  dd. In the Choose Module Type box, select System.ExpressionFilter.  ee. In the Module ID box, type FilterFailure. Click OK.  ff. Click Configure to open the Expression dialog box.  gg. Click Insert.  hh. In the Parameter Name box type Property[@Name='Result'].  ii. In the Operator box select Equals.  jj. In the Value box type Bad.  kk. Click OK to save the expression.  ll. Click OK to save the module configuration.  7. On the Regular tab, do the following:  a. Select Success.  b. Check the Include box next to DataSource.  c. Check the Include box next to FilterSuccess.  d. In the Next Module box next to DataSource select FilterSuccess.  e. In the Next Module box next to FilterSuccess select Monitor State Output.  f. Select Failure.  g. Check the Include box next to DataSource.  h. Check the Include box next to FilterFailure.  i. In the Next Module box next to DataSource select FilterFailure.  j. In the Next Module box next to FilterFailure select Monitor State Output.  8. On the On Demand tab, do the following:  a. Check the box next to Use On Demand Detection.  b. Select Success.  c. Check the Include box next to PassThru.  d. Check the Include box next to Probe.  e. Check the Include box next to FilterSuccess.  f. In the Next Module box next to PassThru select Probe.  g. In the Next Module box next to Probe select FilterSuccess.  h. In the Next Module box next to FilterSuccess select Monitor State Output.  i. Select Failure.  j. Check the Include box next to PassThru.  k. Check the Include box next to Probe.  l. Check the Include box next to FilterFailure.  m. In the Next Module box next to PassThru select Probe.  n. In the Next Module box next to Probe select FilterFailure.  o. In the Next Module box next to FilterFailure select Monitor State Output.  9. On the Configuration Schema tab, do the following:  a. In the Type box next to IntervalSeconds select Integer.  b. Click Add to add a parameter.  c. In the Please enter the requested value box type SyncTime. Click OK.  d. Clear the Required box next to SyncTime.  10. On the Overrideable Parameters tab, do the following:  a. Click Add, then IntervalSeconds.  b. In the Choose a unique identifier box type IntervalSeconds. Click OK.  c. In the Configuration Element box for IntervalSeconds, select Integer.  d. Click Add, then SyncTime.  e. In the Choose a unique identifier box, type SyncTime. Click OK.  11. Click OK to save the module type.  12. Select File, then click Save. |

To create monitor based on custom monitor type

|  |
| --- |
| 1. Select Health Model, then Monitors.  2. In the Monitors pane, expand MyMP.MyComputerRole1 and System.Health.EntityState.  3. Right-click System.Health.AvailabilityState, select New, and then Custom Unit Monitor.  4. In the Choose a unique identifier box, type MyMP.MyTransactionMonitor. Click OK.  5. On the General tab, in the Name box, type My Transaction Monitor.  6. On the Configuration tab, do the following:  a. Click Browse for a type.  b. In the Choose unit monitor type box, select MyMP.MyTransactionMonitorType. Click OK.  c. In the IntervalSeconds box type 900.  d. Clear the text in the ComputerName box. Click the button to the right side of the box, select (Host=Windows Computer), and then Principal Name (Windows Computer).  7. On the Health tab, do the following:  a. In the Health State box for Success select Healthy.  b. In the Health State box for Failure select Critical.  8. On the Alerting tab, do the following:  a. Check the box next to Generate alerts for this monitor.  b. In the Generate an alert when, select The monitor is in a critical or warning health state.  c. In the Alert name: box, type Test transaction failed.  9. Click OK to save the monitor.  10. Select File, then click Save. |

# How to create a monitor based on a Windows PowerShell script

The following procedure shows how to create a monitor based on a custom data source module running a Windows PowerShell script by using Operations Manager 2007 Authoring console. This is the same procedure in [How to create a monitor based on a custom module](#zf0eaf72536184c46bc19db226315f4cc) with the script that is written in Windows PowerShell instead of VBScript.

Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the class to act as the target.

The monitor has the following characteristics:

 Targeted at a class that has only a single instance on an agent. Because there is a single instance, there is no requirement to support cookdown.

 Sets its state based on the comparison of the script value to specified threshold values.

 The monitor supports On Demand Discovery. This requires a probe action module to run the script.

 The script accepts only a single argument for the computer name of the target object’s agent.

 The script itself is only for testing and performs no real function. It simulates a script running a synthetic transaction and returning a status message of success or failure.

Create probe action module to run Windows PowerShell script

|  |
| --- |
| 1. Select Type Library and then Probe Actions.  2. Right-click in the Probe Actions pane and select New and then Composite Probe Action.  3. In the Choose a unique identifier box, type MyMP.ProbeActionModule.MyTransactionPSScript. Click OK.  4. On the General tab, in the Name box, type Transaction PowerShell Script Probe Action.  5. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select Microsoft.Windows.PowerShellPropertyBagProbe.  c. In the Module ID box, type PSScript. Click OK.  d. In the ScriptName box, type MyTransactionScript.ps1.  e. In the TimeoutSeconds box, type 300.  f. Click the Edit button. This starts the external editor.  g. Paste the contents of the following script between the ScriptBody tags in the XML. Replace any text that might already exist.  <![CDATA[  param($computerName)  $testSuccessful = $true  $api = new-object -comObject 'MOM.ScriptAPI'  $api.LogScriptEvent('MyScript.ps1',20,4,$computerName)  $bag = $api.CreatePropertyBag()  $bag.AddValue('ComputerName',$computerName)  if ($testSuccessful -eq $true)  {$bag.AddValue('Result','Good')}  else  {$bag.AddValue('Result','Bad')}  $bag  ]]>  h. Paste the following XML after the </ScriptBody> tag.  <Parameters>  <Parameter>  <Name>ComputerName</Name>  <Value>$Config/ComputerName$</Value>  </Parameter>  </Parameters>  i. Close the external editor to save the script back to the module.  j. Click OK to save the module configuration.  k. In the NextModule column for the PSScript module, select Module Output.  6. On the Configuration Schema tab, do the following:  a. In the Simple Configuration Schema section, click Add to add a parameter.  b. In the Please enter the requested value box, type ComputerName. Click OK.  7. On the Data Types tab, do the following:  a. In the Input Data section, select This module required input data.  b. In the Output Data section, in the Data Type: box select System.PropertyBagData.  8. Click OK to save the module.  9. Select File, and then click Save. |

Create data source module to run probe action on schedule

|  |
| --- |
| 1. Select Type Library and then Data Sources.  2. Right-click in the Data Sources pane and select New and Composite Data Source.  3. In the Choose a unique identifier box, type MyMP.DataSourceModule.MyTransactionPSScriptTimed. Click OK.  4. On the General tab, in the Name box, type Timed Transaction PowerShell Script Data Source  5. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select System.SimpleScheduler.  c. In the Module ID box, type Schedule. Click OK.  d. Click the button to the right side of the IntervalSeconds box and select Promote. This enters the text $Config/IntervalSeconds$  e. Click the button to the right side of the SyncTime box and select Promote. This enters the text $Config/SyncTime$  f. Click OK to save the module configuration.  g. Click Add to add a new module.  h. In the Choose Module Type box, select MyMP.ProbeActionModule.MyTransactionPSScript.  i. In the Module ID box, type Probe. Click OK.  j. Select the button to the right side of the ComputerName box and select Promote. This enters the text $Config/ComputerName$.  k. Click OK to save the module configuration.  l. In the NextModule column for the Schedule module, select Probe.  m. In the NextModule column for the Probe module, select Module Output.  6. On the Configuration Schema tab, do the following:  a. Change the Type for the IntervalSeconds parameter to Integer.  b. Clear the Required box next to the SyncTime parameter. The SyncTime parameter is optional for this module.  7. On the Overrideable Parameters tab, do the following:  a. Click Add, and then click IntervalSeconds.  b. In the Choose a unique identifier box, type IntervalSeconds. Click OK.  c. Change Configuration Element to Integer.  d. Click Add, and then click SyncTime.  e. In the Choose a unique identifier box, type SyncTime. Click OK.  8. On the Data Types tab, make sure that the value in the Data Types box is System.PropertyBagData.  9. Click OK to save the module.  10. Select File, and then click Save. |

Create monitor type using custom data source

|  |
| --- |
| 1. Select Type Library and then Monitor Types.  2. Rightclick in the Monitor Types pane and select New, and then click Composite Monitor Type.  3. In the Choose a unique identifier box, type MyMP.MyPSTransactionMonitorType. Click OK.  4. On the General tab, in the Name box, type My Transaction PowerShell Script Monitor Type.  5. On the States tab, do the following:  a. Select 2 State Monitor Type.  b. In the ID of state 1 box, type Success.  c. In the ID of state 2 box, type Failure.  6. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select System.PassThroughProbe.  c. In the Module ID box, type PassThru. Click OK.  d. Click OK to save the module configuration.  e. Click Add to add a module.  f. In the Choose Module Type box, select MyMP.ProbeActionModule.MyTransactionPSScript.  g. In the Module ID box, type Probe. Click OK.  h. Click the button to the right side of the ComputerName box and select Promote. This enters the text $Config/ComputerName$  i. Click OK to save the module configuration.  j. Click Add to add a new module.  k. In the Choose Module Type box, select MyMP.DataSourceModule.MyTransactionPSScriptTimed.  l. In the Module ID box, type DataSource. Click OK.  m. Click the button to the right side of the IntervalSeconds box and select Promote . This enters the text $Config/IntervalSeconds$  n. Click the button to the right side of the ComputerName box and select Promote. This enters the text $Config/ComputerName$  o. Click Edit to start the custom editor. The SyncTime parameter must be added manually with an XML editor. This is because the Authoring Console only populates the Configuration dialog box by using required parameters for the selected module. The SyncTime parameter was configured not to be required.  p. After the line  <IntervalSeconds>$Config/IntervalSeconds$</IntervalSeconds>  Add the following line  <SyncTime>$Config/SyncTime$</SyncTime>  q. Close the editor to add the XML back the Authoring Console.  Note  If you receive an error that says that the IntervalSeconds parameter is invalid according to its data type, click Ignore. This error message occurs because the IntervalSeconds parameter is configured as an integer, and the Authoring Console is reading $Config/IntervalSeconds$ as a string. This variable will be replaced with an integer value when the workflow is run so the error can be ignored.  r. Click OK to save the module configuration.  s. Click Add to add a new module.  t. In the Choose Module Type box, select System.ExpressionFilter.  u. In the Module ID box, type FilterSuccess. Click OK.  v. Click Configure to open the Expression dialog box.  w. Click Insert.  x. In the Parameter Name box type Property[@Name='Result'].  y. In the Operator box select Equals.  z. In the Value box type Good.  aa. Click OK to save the expression.  bb. Click OK to save the module configuration.  cc. Click Add to add a new module.  dd. In the Choose Module Type box, select System.ExpressionFilter.  ee. In the Module ID box, type FilterFailure. Click OK.  ff. Click Configure to open the Expression dialog box.  gg. Click Insert.  hh. In the Parameter Name box type Property[@Name='Result'].  ii. In the Operator box select Equals.  jj. In the Value box type Bad.  kk. Click OK to save the expression.  ll. Click OK to save the module configuration.  7. On the Regular tab, do the following:  a. Select Success.  b. Check the Include box next to DataSource.  c. Check the Include box next to FilterSuccess.  d. In the Next Module box next to DataSource select FilterSuccess.  e. In the Next Module box next to FilterSuccess select Monitor State Output.  f. Select Failure.  g. Check the Include box next to DataSource.  h. Check the Include box next to FilterFailure.  i. In the Next Module box next to DataSource select FilterFailure.  j. In the Next Module box next to FilterFailure select Monitor State Output.  8. On the On Demand tab, do the following:  a. Check the box next to Use On Demand Detection.  b. Select Success.  c. Check the Include box next to PassThru.  d. Check the Include box next to Probe.  e. Check the Include box next to FilterSuccess.  f. In the Next Module box next to PassThru select Probe.  g. In the Next Module box next to Probe select FilterSuccess.  h. In the Next Module box next to FilterSuccess select Monitor State Output.  i. Select Failure.  j. Check the Include box next to PassThru.  k. Check the Include box next to Probe.  l. Check the Include box next to FilterFailure.  m. In the Next Module box next to PassThru select Probe.  n. In the Next Module box next to Probe select FilterFailure.  o. In the Next Module box next to FilterFailure select Monitor State Output.  9. On the Configuration Schema tab, do the following:  a. In the Type box next to IntervalSeconds select Integer.  b. Click Add to add a parameter.  c. In the Please enter the requested value box type SyncTime. Click OK.  d. Clear the Required box next to SyncTime.  10. On the Overrideable Parameters tab, do the following:  a. Click Add and then IntervalSeconds.  b. In the Choose a unique identifier box type IntervalSeconds. Click OK.  c. In the Configuration Element box for IntervalSeconds select Integer.  d. Click Add and then SyncTime.  e. In the Choose a unique identifier box type SyncTime. Click OK.  11. Click OK to save the module type.  12. Select File, and then click Save. |

Create monitor based on custom monitor type

|  |
| --- |
| 1. Select Health Model and then Monitors.  2. In the Monitors pane, expand MyMP.MyComputerRole1 and then System.Health.EntityState.  3. Right-click System.Health.AvailabilityState and select New. and then select Custom Unit Monitor.  4. In the Choose a unique identifier box, type MyMP.MyPSTransactionMonitor. Click OK.  5. On the General tab, in the Name box, type My PowerShell Transaction Monitor.  6. On the Configuration tab, do the following:  a. Click Browse for a type.  b. In the Choose unit monitor type box select MyMP.MyPSTransactionMonitorType. Click OK.  c. In the IntervalSeconds box type 900.  d. Clear the text in the ComputerName box. Click the button to the right side of the box and select (Host=Windows Computer) and then Principal Name (Windows Computer).  7. On the Health tab, do the following:  a. In the Health State box for Success select Healthy.  b. In the Health State box for Failure select Critical.  8. On the Alerting tab, do the following:  a. Check the box next to Generate alerts for this monitor.  b. In the Generate an alert when select The monitor is in a critical health state.  c. In the Alert name: box type Test PowerShell transaction failed.  9. Click OK to save the monitor.  10. Select File, and then click Save. |

# How to create a monitor and rule that share a script supporting cookdown

The following procedure shows how to create a monitor and rule based on a common data source module running a script using Operations Manager 2007 Authoring console. Before you perform this procedure, you must first complete the prerequisite procedure [How to Create a Class](#z6554b8ab6eb34b5eb62d830283a06596) in which you create the class to act as the target.

The workflows have the following characteristics:

 Rule and monitor targeted at a class that has multiple instances on an agent. Because there are multiple instances, cookdown support is required.

 The rule collects a performance value from the script while the monitor sets its state based on the comparison of the script value to specified threshold values.

 The monitor does not support On Demand Discovery so that it does not require that you use a probe action module to run the script.

 The script accepts arguments for the computer name of the target object’s agent and the version of the application.

 The script itself is only a test that generates a constant value.

To create data source module to run script on schedule

|  |
| --- |
| 1. Select Type Library, then Data Sources.  2. Right-click the Data Sources pane and select New and Composite Data Source.  3. In the Choose a unique identifier box, type MyMP.DataSourceModule.MyPerformanceScriptTimed. Click OK.  4. On the General tab, in the Name box, type Timed Performance Script Data Source  5. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select Microsoft.Windows.TimedScript.PropertyBagProvider.  c. Click the button to the right side of the IntervalSeconds box and select Promote. This will enter the text $Config/IntervalSeconds$  d. In the Module ID box, type Script. Click OK.  e. Click the button to the right side of the SyncTime box and select Promote. This will enter the text $Config/SyncTime$  f. In the ScriptName box, type MyCookdownScript.vbs.  g. In the Arguments box, type $Config/ComputerName$ $Config/Version$.  h. In the TimeoutSeconds box, type 300.  i. Click the button. This starts the custom editor. Edit  j. Paste the complete contents of the following script between the ScriptBody tags in the XML. Replace any text that might already exist.  <![CDATA[  sComputerName = WScript.Arguments(0)  sVersion = WScript.Arguments(1)  Set oAPI = CreateObject("MOM.ScriptAPI")  oAPI.LogScriptEvent "MyCookdownScript.vbs",10,4, "Running script on " & sComputerName & ". Version is " & sVersion  For i = 1 to 3  Set oBag = oAPI.CreatePropertyBag()  Call oBag.AddValue("ComputerName",sComputerName)  Call oBag.AddValue("ComponentName","Component" & i)  Call oBag.AddValue("Value",30)  oAPI.AddItem(oBag)  Next  oAPI.ReturnItems  ]]>  k. Close the editor to save the script back to the module.  l. If you receive an error that says that the IntervalSeconds parameter is invalid according to its data type, click Ignore. This error message occurs because the IntervalSeconds parameter is configured as an integer, and the Authoring Console is reading $Config/IntervalSeconds$ as a string. This variable will be replaced with an integer value when the workflow is run so the error can be ignored.  m. Click OK to save the module configuration.  n. Click Add to add a new module.  o. In the Choose Module Type box, select System.Performance.DataGenericMapper.  p. In the Module ID box, type MapToPerf. Click OK.  q. In the ObjectName box, type MyApp.  r. In the CounterName box, type TestCounter.  s. In the InstanceName box, type $Data/Property[@Name='ComponentName']$  t. In the Value box, type $Data/Property[@Name='Value']$.  u. Click OK.  v. In the NextModule column for the Script module, select MapToPerf.  w. In the NextModule column for the MapToPerf module, select Module Output.  6. On the Configuration Schema tab, do the following:  a. Change the Type for the IntervalSeconds parameter to Integer.  b. Clear the Required box next to the SyncTime parameter. The SyncTime parameter is optional for this module.  c. In the Simple Configuration Schema section, click Add to add a parameter.  d. In the Please enter the requested value box, type ComputerName. Click OK.  e. In the Simple Configuration Schema section, click Add to add a parameter.  f. In the Please enter the requested value box, type Version. Click OK.  7. On the Overrideable Parameters tab, do the following:  a. Click Add, then IntervalSeconds.  b. In the Choose a unique identifier box, type IntervalSeconds. Click OK.  c. Change Configuration Element to Integer.  d. Click Add, then SyncTime.  e. In the Choose a unique identifier box, type SyncTime. Click OK.  8. On the Data Types tab ensure the value in the Data Types box is System.Performance.Data.  9. Click OK to save the module.  10. Select File, then click Save. |

To create data source module to filter on instance

|  |
| --- |
| 1. Select Type Library, then Data Sources.  2. Right-click the Data Sources pane and select New and Composite Data Source.  3. In the Choose a unique identifier box, type MyMP.DataSourceModule.MyPerformanceScriptFiltered. Click OK.  4. On the General tab, in the Name box, type Filtered Performance Script Data Source  5. On the Member Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select MyMP.DataSourceModule.MyPerformanceScriptTimed.  c. In the Module ID box, type Script. Click OK.  d. Click the button to the right side of the IntervalSeconds box and select Promote. This will enter the text $Config/IntervalSeconds$  e. Click the button to the right side of the ComputerName box and select Promote. This will enter the text $Config/ComputerName $  f. Click the button to the right side of the Version box and select Promote. This will enter the text $Config/Version$  g. Click Edit to start the custom editor. The SyncTime parameter must be added manually with an XML editor. This is because the Authoring Console only populates the Configuration dialog box by using parameters required for the selected module. The SyncTime parameter was configured to not be required.  h. After the line  <IntervalSeconds>$Config/IntervalSeconds$</IntervalSeconds>  Add the following line  <SyncTime>$Config/SyncTime$</SyncTime>  i. Close the editor to save the script back to the module.  Note  If you receive an error saying that the IntervalSeconds parameter is invalid according to its data type, click Ignore. This error message occurs because the IntervalSeconds parameter is configured as an integer, and the Authoring Console is reading $Config/IntervalSeconds$ as a string. This variable will be replaced with an integer value when the workflow is run so the error can be ignored.  j. Click OK to save the module configuration.  k. Click Add to add a module.  l. In the Choose Module Type box, select System.ExpressionFilter.  m. In the Module ID box, type FilterComponent. Click OK.  n. Click Configure to open the Expression dialog box.  o. Click Insert.  p. In the Parameter Name box type InstanceName.  q. In the Operator box select Equals.  r. In the Value box type $Config/ComponentName$.  s. Click OK to save the expression.  t. Click OK to save the module configuration.  u. In the NextModule column for the Script module, select FilterComponent.  v. In the NextModule column for the FilterComponent module, select Module Output.  6. On the Configuration Schema tab, do the following:  a. Change the Type for the IntervalSeconds parameter to Integer.  b. In the Simple Configuration Schema section, click Add to add a parameter.  c. In the Please enter the requested value box, type SyncTime. Click OK.  d. Clear the Required box next to the SyncTime parameter. The SyncTime parameter is optional for this module.  e. In the Simple Configuration Schema section, click Add to add a parameter.  f. In the Please enter the requested value box, type ComponentName. Click OK.  7. On the Overrideable Parameters tab, do the following:  a. Click Add, then IntervalSeconds.  b. In the Choose a unique identifier box, type IntervalSeconds. Click OK.  c. Change Configuration Element to Integer.  d. Click Add, then SyncTime.  e. In the Choose a unique identifier box, type SyncTime. Click OK.  8. On the Data Types tab ensure the value in the Data Types box is System.Performance.Data.  9. Click OK to save the module.  10. Select File, then click Save. |

To create rule to collect performance data

|  |
| --- |
| 1. Select Health Model, then Rules.  2. Right-click the Rules pane and select New, then Custom Rule.  3. In the Choose a unique identifier box, type MyMP. MyAppComponent.PerformanceCollectionRule. Click OK.  4. On the General tab, do the following:  a. In the Name box type Collect MyApp Component performance data.  b. In the Target box, select MyMP.MyApplicationComponent.  5. On the Modules tab, do the following:  a. In the Data Sources section click Create.  b. In the Choose Module Type box, select MyMP.DataSourceModule.MyPerformanceScriptFiltered.  c. In the Module ID box, type DS. Click OK.  d. Click Edit to modify the module parameters.  e. In the IntervalSeconds box, type 900.  f. Clear the text in the ComputerName box. Click the button to the right of the box, select (Host=My Computer Role 1). Now select (Host=Windows Computer) and then Principal Name (Windows Computer). This enters the text $Target/Host/Host/Property[Type="Windows!Microsoft.Windows.Computer"]/PrincipalName$.  g. Clear the text in the Version box. Click the button to the right side of the box and select (Host=My Computer Role 1) and then Version (My Computer Role Base). This will enter the text $Target/Host/Property[Type="MyMP.MyComputerRoleBase"]/Version$.  h. Clear the text in the ComponentName box. Click the button to the right of the box and select ComponentName.  i. Click OK to save the module configuration.  j. In the Actions section click Create.  k. In the Choose Module Type box, select Microsoft.SystemCenter.CollectPerformanceData.  l. In the Module ID box, type WriteToDB. Click OK.  m. In the Actions section click Create.  n. In the Choose Module Type box, select Microsoft.SystemCenter.DataWarehouse.PublishPerformanceData.  o. In the Module ID box, type WriteToDW. Click OK.  6. Click OK.  7. Select File, then click Save. |

To create monitor type using custom data source

|  |
| --- |
| 1. Select first Type Library and then select Monitor Types.  2. Right-click in the Monitor Types pane, select New, and then Composite Monitor Type.  3. In the Choose a unique identifier box, type MyMP.MyAppComponentMonitorType. Click OK.  4. On the General tab, do the following:  a. In the Name box, type MyApplicationComponent Monitor Type.  5. On the States tab, do the following:  a. Select 3 State Monitor Type.  b. In the ID of state 1 box type UnderThreshold.  c. In the ID of state 2 box type OverWarningThreshold.  d. In the ID of state 3 box type OverErrorThreshold.  6. On the Modules tab, do the following:  a. Click Add to add a module.  b. In the Choose Module Type box, select MyMP.DataSourceModule.MyScriptFiltered.  c. In the Module ID box, type DataSource. Click OK.  d. Click the button to the right of the IntervalSeconds box and select Promote. This enters the text $Config/IntervalSeconds$  e. Click the button to the right side of the ComputerName box and select Promote. This enters the text $Config/ComputerName$  f. Click the button to the right side of the Version box and select Promote. This enters the text $Config/Version$  g. Click the button to the right side of the ComponentName box and select Promote. This enters the text $Config/ComponentName $  h. Click Edit to start the custom editor. The SyncTime parameter must be added manually with an XML editor. This is because the Authoring console only populates the Configuration dialog box with required parameters for the selected module. The SyncTime parameter was configured to not be required.  i. After the line  <IntervalSeconds>$Config/IntervalSeconds$</IntervalSeconds>  Add the following line  <SyncTime>$Config/SyncTime$</SyncTime>  j. Close the editor to add the XML back the Authoring Console.  Note  If you receive an error saying that the IntervalSeconds parameter is invalid according to its data type, click Ignore. This error message occurs because the IntervalSeconds parameter is configured as an integer, and the Authoring Console is reading $Config/IntervalSeconds$ as a string. This variable will be replaced with an integer value when the workflow is run so the error can be ignored.  k. Click OK to save the module configuration.  l. Click Add to add a module.  m. In the Choose Module Type box, select System.ExpressionFilter.  n. In the Module ID box, type FilterUnderThreshold. Click OK.  o. Click Configure to open the Expression dialog box.  p. Click Insert.  q. In the Parameter Name box type Value.  r. In the Operator box select Less than.  s. In the Value box type $Config/WarningThreshold$.  t. Click OK.  u. In each >>>@Type box, change the value String to Integer.  v. Click OK to save the module configuration.  w. Click Add to add a module.  x. In the Choose Module Type box, select System.ExpressionFilter.  y. In the Module ID box, type FilterOverWarningThreshold. Click OK.  z. Click Configure open the Expression dialog box.  aa. Click Insert.  bb. In the Parameter Name box type Value.  cc. In the Operator box select Greater than or equal to.  dd. In the Value box type $Config/WarningThreshold$. Click OK.  ee. Click Insert.  ff. In the Parameter Name box type Value.  gg. In the Operator box select Less than.  hh. In the Value box type $Config/ErrorThreshold$. Click OK.  ii. In each >>>@Type box, change the value String to Integer.  jj. Click OK to save the module configuration.  kk. Click Add to add a module.  ll. In the Choose Module Type box, select System.ExpressionFilter.  mm. In the Module ID box, type FilterOverErrorThreshold. Click OK.  nn. Click Configure.  oo. Click Insert.  pp. In the Parameter Name box type Value.  qq. In the Operator box select Greater than or equal to.  rr. In the Value box type $Config/ErrorThreshold$.  ss. Click OK.  tt. In each >>>@Type box, change the value String to Integer.  uu. Click OK to save the module configuration.  7. On the Regular tab, do the following:  a. Select UnderThreshold.  b. Check the Include box next to DataSource.  c. Check the Include box next to FilterUnderThreshold.  d. In the Next Module box next to DataSource select FilterUnderThreshold.  e. In the Next Module box next to FilterUnderThreshold select Monitor State Output.  f. Select OverWarningThreshold.  g. Check the Include box next to DataSource.  h. Check the Include box next to FilterOverWarningThreshold.  i. In the Next Module box next to DataSource select FilterOverWarningThreshold.  j. In the Next Module box next to FilterOverWarningThreshold select Monitor State Output.  k. Select OverErrorThreshold.  l. Check the Include box next to DataSource.  m. Check the Include box next to FilterOverErrorThreshold.  n. In the Next Module box next to DataSource select FilterOverErrorThreshold.  o. In the Next Module box next to FilterOverErrorThreshold select Monitor State Output.  8. On the Configuration Schema tab, do the following:  a. In the Type box next to IntervalSeconds select Integer.  b. Click Add to add a parameter.  c. In the Please enter the requested value box type SyncTime. Click OK.  d. Clear the Required box next to SyncTime.  e. Click Add to add a parameter.  f. In the Please enter the requested value box type WarningThreshold. Click OK.  g. In the Type box next to WarningThreshold select Integer.  h. Click Add to add a parameter.  i. In the Please enter the requested value box type ErrorThreshold. Click OK.  j. In the Type box next to ErrorThreshold select Integer.  9. On the Overrideable Parameters tab, do the following:  a. Click Add and then IntervalSeconds.  b. In the Choose a unique identifier box type IntervalSeconds.  c. In the Configuration Element box for IntervalSeconds select Integer.  d. Click Add and then SyncTime.  e. In the Choose a unique identifier box type SyncTime.  f. Click Add, then WarningThreshold.  g. In the Choose a unique identifier box type WarningThreshold.  h. In the Configuration Element box for WarningThreshold select Integer.  i. Click Add, then ErrorThreshold.  j. In the Choose a unique identifier box type ErrorThreshold.  k. In the Configuration Element box for ErrorThreshold select Integer.  10. Click OK to save the monitor type.  11. Select File, then click Save. |

To create monitor based on custom monitor type

|  |
| --- |
| 1. Select Health Model, then Monitors.  2. Expand MyMP.MyApplicationComponent and then System.Health.EntityState.  3. Right-click System.Health.PerformanceState, select New, and then Custom Unit Monitor.  4. In the Choose a unique identifier box, type MyMP.MyAppComponent.PerformanceMonitor. Click OK.  5. On the General tab, in the Name box, type My Application Component Performance.  6. On the Configuration tab, do the following:  a. Click Browse for a type….  b. In the Choose unit monitor type box select MyMP.MyAppComponentMonitorType.  c. In the IntervalSeconds box type 900.  d. Clear the text in the ComputerName box. Click the button to the right side of the box and select (Host=My Computer Role 1) and then(Host=Windows Computer) and then Principal Name (Windows Computer).  e. Clear the text in the Version box. Click the button to the right side of the box and select (Host=My Computer Role 1) and then Version (My Computer Role Base).  f. Clear the text in the ComponentName box. Click the button to the right side of the box and select ComponentName.  g. In the WarningThreshold box type 10.  h. In the ErrorThreshold box type 20.  7. On the Health tab, do the following:  a. In the Health State box for UnderThreshold select Healthy.  b. In the Health State box for OverWarningThreshold select Warning.  c. In the Health State box for OverErrorThreshold select Critical.  8. On the Alerting tab, do the following:  a. Check the box next to Generate alerts for this monitor.  b. In the Generate an alert when… select The monitor is in a critical or warning health state.  c. In the Alert name: box type Performance problem detected.  9. Click OK to save the monitor.  10. Select File, then click Save. |