MICRO FOCUS

Answer Server

Software Version: 11.6

Administration Guide

Document Release Date: February 2018

Software Release Date: February 2018

Legal notices

Warranty

The only warranties for Seattle SpinCo, Inc. and its subsidiaries ("Seattle") products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Seattle shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Restricted rights legend

Confidential computer software. Except as specifically indicated, valid license from Seattle required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright notice

Trademark notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Documentation updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To verify you are using the most recent edition of a document, go to https://softwaresupport.softwaregrp.com/group/softwaresupport/search-result?doctype=online help.

This site requires you to sign in with a Software Passport. You can register for a Passport through a link on the site.

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your Micro Focus sales representative for details.

Support

Visit the Micro Focus Software Support Online website at https://softwaresupport.softwaregrp.com.

This website provides contact information and details about the products, services, and support that Micro Focus offers.

Micro Focus online support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support website to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Access the Software Licenses and Downloads portal
- Download software patches
- Access product documentation
- Manage support contracts

- Look up Micro Focus support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require you to register as a Passport user and sign in. Many also require a support contract.

You can register for a Software Passport through a link on the Software Support Online site.

To find more information about access levels, go to https://softwaresupport.softwaregrp.com/web/softwaresupport/access-levels.

Contents

Part I: Get Started With Answer Server 9
Chapter 1: Introduction 11
Answer Server System Architecture 11
Answer Bank 12
Fact Bank 12
Passage Extractor 13
Conversation 13
OEM Certification 14
Chapter 2: Install and Set Up Answer Server 15
Install Answer Server 15
External Dependencies 16
Install an IDOL Component as a Service on Windows 16
Install an IDOL Component as a Service on Linux 17
Install a Component as a Service for a systemd Boot System 18
Install a Component as a Service for a System V Boot System 19
Licenses 19
Display License Information 20
Configure the License Server Host and Port 21
Revoke a Client License 21
Troubleshoot License Errors 22
Chapter 3: Configure Answer Server 25
General Configuration 25
Configure Answer Server Systems 26
Configure an Answer Bank System 26
Configure a Fact Bank System 26
Configure a Passage Extractor System 27
Configure a Conversation System 27
Language Configuration 27
Answer Bank Language Configuration 28
Fact Bank and Passage Extractor Language Configuration 28
Configure Client Authorization 29
Configure SSL Communication Between Components 30
Customize Logging 31
Validate the Configuration File 32
Chapter 4: Run Answer Server 33
Start Answer Server 33
Stop Answer Server 33
Verify that Answer Server is Running 34
GetStatus 34
GetLicenseInfo 34

Send Actions to Answer Server 35

Part II: Configure Answer Server Systems 37

Chapter 5: Set Up an Answer Bank System 39
Configure the Answer Bank System 39
Configure the Answer Bank Agentstore 40
Manage an Answer Bank 40
Find the JSON Schema for Your Update 41
Add a Question 41
Find the Likelihood of Existing Answers 41
Find the Likely Answers to a Question 42
Question Equivalence Rules 42
Special Rule Types 43
Generate a Question Equivalence Rule 44
Test Your Question Equivalence Rule 45
Find Questions that Do Not Match the Rule 45
Find Questions in Other Classes that Match the Rule 45
Test Whether Questions Match a Specified Rule 46
Create a Question Equivalence Class and Add an Answer 47
Update a Question Equivalence Class 48
Update the Question State 50
Automatic Question State Updates 50
Update the Question State Manually 50
Delete a Question or Question Equivalence Class 51
Undelete a Question or Question Equivalence Class 52
Modify the Expiration Time 52
Check the Status of an Update 53
Store Statistics for Your Answer Bank 53
Retrieve the Information Stored in an Answer Bank 54
GetResources 55
GetStats 55

Chapter 6: Set Up a Fact Bank System 57
Configure the Fact Bank System 57
Configure a Fact Bank with a SQL Database Fact Store 58
Configure a Fact Bank to Call a Lua Script 59
Configure the Fact Store 60
Set Up a SQL Backend as Fact Store 60
Manage Your Tables 61
Facts Table 61
Qualifiers Table 62
Sources Table 62
Security_Types Table 63
SQL Fact Store Example 63
Use a Lua Scripts to Retrieve Facts 65
Create a Fact Retrieval Script 66

Create Coding Files 67
Example Data 67
Generate the Property Code Files 67
Generate the Entity Code Files 68
Generate the Fact Store Data 69
Create a Fact Store Table for a SQL Database 69
The Question Parser Eduction Grammar 69
Processors 70
Example Questions 70
Modify the Question Parser Eduction Grammar 72
Configure Security in Fact Bank 73
Configure the Security Types in Answer Server 73
Set Up Fact Store Tables for Security 74
Chapter 7: Set Up a Passage Extractor System 75
Configure the Passage Extractor System 75
Train Passage Extractor Classifiers 76
Create a Training File 77
Train a Classifier 77
Classifier Behavior File 78
Training File Labels 78
Entity Extraction in Passage Extractor 81
Configure the Passage Extractor Agentstore 81
Configure the Agentstore Component 81
Index Entity Agents 82
Customize Entity Extraction 82
The Entity Extraction File Format 83
Modify the Entity Extraction File 85
Use a Different Entity Extraction File 85
Troubleshoot Passage Extractor 86
Chapter 8: Ask Questions in Answer Server 87
Ask a Question 87
Chapter 9: Set Up a Conversation System 89
Configure the Conversation System 89
Create a Task Configuration File 90
Pre-Task Actions 91
Conversation Triggers 92
Agent Triggers 93
Regular Expressions Triggers 94
Simple Triggers 95
Task Disambiguation 97
Trigger Options 98
Task Requirements 98
Automatic Requirement Gathering 101
Response Validation 103
Simple Validation 104

Regular Expression Validator ... 105
Eduction Validator ... 105
Lua Validator ... 106
Process Non-Valid Input ... 107
Post-Task Actions ... 108
Task Routing ... 110
Configure Simple Routing ... 110
Configure Conditional Routing ... 111
Use Routing in a Lua Function ... 113
Lua Processing Scripts ... 113
Default Tasks ... 115
Initial Task ... 115
Fallback Task ... 115
Default Messages ... 116
Response for Non-Valid Input ... 116
Response for Disambiguation ... 116
Response for Multiple Answer Disambiguation ... 117
Task Cancellation ... 120
User Cancellation ... 120
System Cancellation ... 121
Task Configuration Example ... 122
Configure the Conversation Agentstore ... 122
Configure the Agentstore Component ... 122
Configure the Conversation System to Use Agentstore ... 123
Index Conversation Trigger Agents ... 123
Chapter 10: Hold Conversations in Answer Server ... 125
Hold a Conversation ... 125
Create a Conversation Session ... 125
Retrieve Active Conversation Sessions ... 126
Start and Continue a Conversation ... 126
Close a Conversation Session ... 126
Retrieve a Conversation Transcript ... 127
Chapter 10: Use Natural Language Generation in Answer Server ... 127
Configure Natural Language Generation ... 127
Run Natural Language Generation ... 128
Part III: Appendixes ... 129
Appendix A: Debug Your Conversation Lua Scripts ... 131
Glossary ... 133
Send documentation feedback ... 135

Part I: Get Started With Answer Server

This section introduces Micro Focus Answer Server and describes how to install and run the server.

- Introduction, on page 11
- Install and Set Up Answer Server, on page 15
- Configure Answer Server, on page 25
- Run Answer Server, on page 33

Chapter 1: Introduction

Micro Focus Answer Server uses IDOL technology to provide specific and concise answers to user questions.

In a traditional IDOL Server system, the user provides some search terms, or uses special search syntax, and the server returns a list of related documents. In Answer Server, the user specifies a question, and the server returns as specific an answer as possible.

The Answer Server has four types of system to answer different question types.

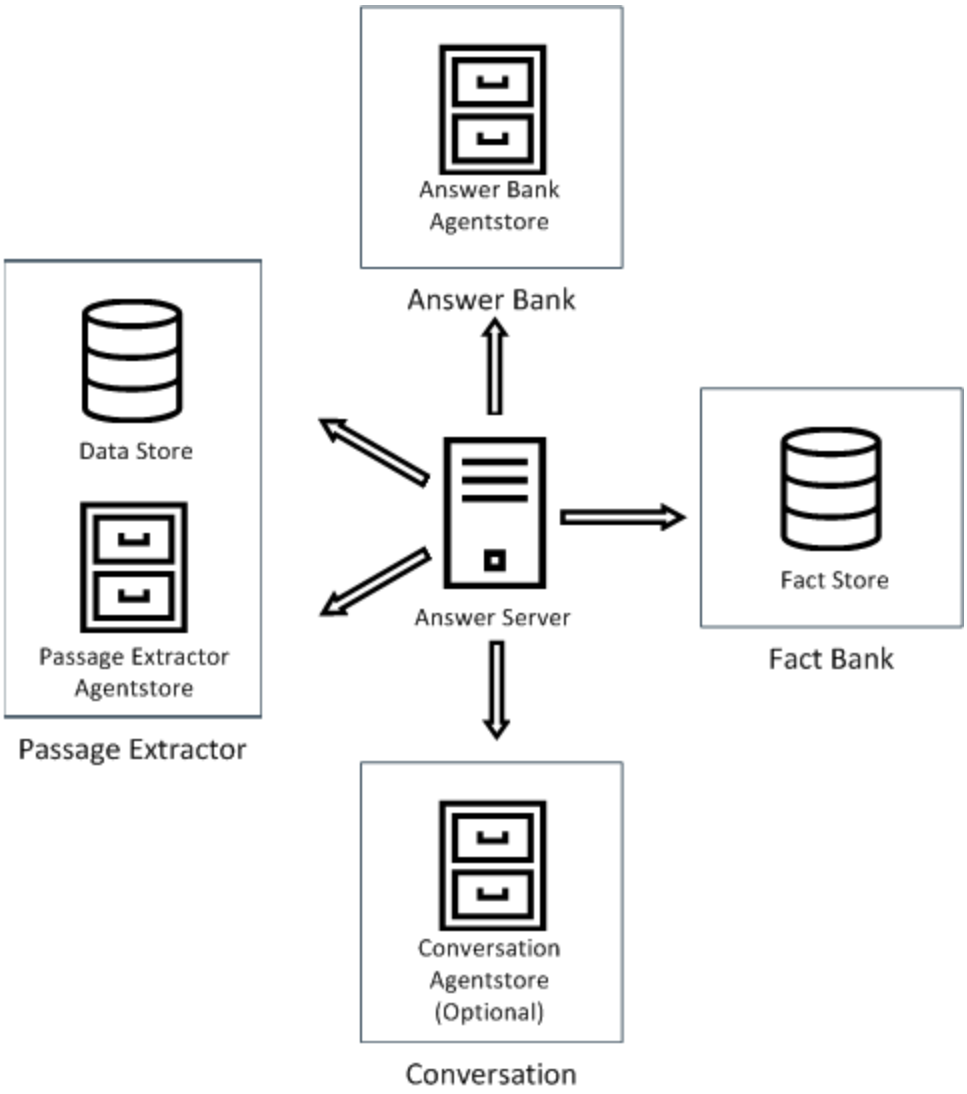
- **Answer Bank**. The Answer Bank contains a store of reference questions and answers, which you can add and administer. When a user asks a question, the Answer Bank queries the store of existing questions for any that match, and returns the relevant answers. You can use an Answer Bank to maintain an FAQ list to answer questions such as:
  - How do I fix my phone screen?
  - What does error 404 mean?
  - Can I use my phone to send photos of cats?
- **Fact Bank**. The Fact Bank contains a store of factual information, to return simple factual answers. For example, you could use a Fact Bank to answer questions such as:
  - What is the population of the USA?
  - Who is the CEO of Micro Focus?
  - What is the average June temperature in Antarctica?
- **Passage Extractor**. The Passage Extractor links to a store of documents that contain general information that might be useful for answering questions (for example, your normal data IDOL Server). When a user asks a question, the Passage Extractor queries the document store and attempts to extract short sentences or paragraphs that contain relevant answers. You can use a Passage Extractor to answer general questions that you do not have a fact store for, or that do not have a simple answer, such as:
  - What is the tallest building in the world?
  - What did the only repealed amendment to the US Constitution deal with?
  - What is the weight of the Eiffel Tower?
- **Conversation**. The Conversation module runs a real time conversation task with your end users. It allows you to set up an interactive virtual assistant to answer common user queries.

You can configure as many different versions of each system as you need. When you send a question to Answer Server, you can specify which of the configured systems you want to retrieve answers from.

The following sections describe the setup for these systems in more detail.

Answer Server System Architecture

The following diagram shows the different components of the Answer Server system.



Answer Bank

The answer bank system uses a dedicated IDOL Agentstore component.

The Agentstore is a specially configured IDOL Content component that stores the set of questions and their answers. You can also create *question equivalence classes*, which store a set of equivalent questions that map to the same answer.

Fact Bank

The fact bank system consists of three pieces:

- Fact Store (SQL database or Lua script). The fact store contains the factual information that you want to retrieve. Answer Server uses the parsed question and the associated entity and property codes to search the fact store for relevant facts. Usually the fact store component is a

SQL database. Alternatively, you can use a Lua script to retrieve facts from an external fact store.

- Question Parser (IDOL Eduction). The fact bank uses a specialized Eduction grammar to parse questions and extract the parts of the question that define the fact that the question requests. For example, for the question "what is the population of the USA", Eduction determines that the user wants to find the *population* property of the *USA* entity. The Eduction module is embedded in Answer Server, so you do not need to install a separate component.
- Coding files. The coding files map entities, properties, and their synonyms to a unique code. Answer Server uses this code to retrieve data from the fact store. For example, the coding files can store the different ways of referring to the country *USA* as a single code (*United States of America*, *US*, and so on). The code can be easily retrieved to match to associated facts.

The fact bank system also includes additional Eduction grammars for advanced time normalization. Advanced time normalization extracts dates and times in various formats from questions and normalize them to a consistent format, to improve fact retrieval.

Passage Extractor

The passage extractor system consists of two components:

- Data Store (IDOL Content component). The data store is an IDOL Content component that contains the documents that you want to attempt to extract answers from. This might be a general knowledge base (such as an index of Wikipedia documents), or a data set that is more specific to your business use (such as an index of your company policy documents). This data store does not have to be unique to Answer Server. That is, you can use your normal IDOL Server data Content component.
- Passage Extractor Agentstore. The passage extractor uses Agentstore for some of the entity extraction operations. For example, you can set up agents that define people and place entities, and the passage extractor uses agent queries to extract those entities from the documents that you use to find answers. Agent matching is often quicker than Eduction entity matching for simple entities that consist of fixed text, such as names.

The passage extractor also requires:

- Eduction grammars. The passage extractor uses the Question Parser Eduction grammar, as well as grammars for extracting entities from passages that might contain answers, and for identifying sentences and paragraphs that might form legitimate answers, by using pattern matching. The Eduction module is embedded in Answer Server, so you do not need to install a separate component.
- Classifier training files. These files define types of questions, which determine the type of answer it looks for. You can configure Answer Server to save the training files and training data.

Conversation

The conversation system does not have any required subcomponents. You configure conversation tasks by using a JSON configuration file, which describes the task, including:

- introductory text.
- the *triggers* to use to activate a particular conversation or option.

- routing information to describe how to proceed through the conversation task, depending on user input.
- details of lua scripts to run at particular points in the conversation to parse user input or provide more detailed conversation processing.

You can set your conversation triggers by using fixed phrases, regular expressions, or IDOL agents. If you want to use IDOL agents as triggers, you must configure an IDOL Agentstore component.

OEM Certification

Answer Server works in OEM licensed environments.

Chapter 2: Install and Set Up Answer Server

This section describes how to install Answer Server.

- Install Answer Server 15
  - External Dependencies 16
- Install an IDOL Component as a Service on Windows 16
- Install an IDOL Component as a Service on Linux 17
  - Install a Component as a Service for a systemd Boot System 18
  - Install a Component as a Service for a System V Boot System 19
- Licenses 19
  - Display License Information 20
  - Configure the License Server Host and Port 21
  - Revoke a Client License 21
  - Troubleshoot License Errors 22

Install Answer Server

Answer Server is available as a ZIP package which includes the files you need to set up Answer Server.

The Answer Server ZIP package includes:

- the Answer Server executable file.
- the Answer Server configuration file.

It also includes the appropriate data and configuration files for the different answer systems.

For Answer Bank, the ZIP package includes:

- a configuration file for the Answer Bank Agentstore component

For Fact Bank, the ZIP package includes:

- the Fact Bank Eduction grammar ECR and XML files
- the Fact Bank Eduction Lua scripts
- a set of example files for setting up a SQL fact store
- a simple example script for creating a Lua fact store
- a configuration file for an IDOL Content component Fact Store (the IDOL Content component fact store option is deprecated in Answer Server version 11.5 and later. See Configure the Fact Store, on page 60)

For Passage Extractor, the ZIP package includes:

- a configuration file for the Passage Extractor Agentstore component
- sample entity IDX files for the Passage Extractor Agentstore entity extraction

- the classifier and label files for the passage extractor question classifiers
- the `entity_extraction.json` and `surface_patterns.json` configuration files
- the Passage Extractor Eduction grammar, and several entity extraction Eduction grammar ECR files

To install Answer Server, download and unzip the package. You can optionally install Answer Server as a service. See Install an IDOL Component as a Service on Windows, below or Install an IDOL Component as a Service on Linux, on the next page.

The ZIP package does not include the subcomponent executable files. For each system, you must also download and install the associated subcomponents. The following sections provide more information about the requirements for each system:

- Set Up an Answer Bank System, on page 39
- Set Up a Fact Bank System, on page 57
- Set Up a Passage Extractor System, on page 75

External Dependencies

On Linux operating system platforms, if you want to use any of the functionality that uses an ODBC connection, Micro Focus recommends explicitly setting the `ODBCSYSINI` environment variable. This variable specifies the directory that contains the `odbcinst.ini` file, which contains details of available ODBC drivers on your system.

For example, on some systems you must set `ODBCSYSINI=/etc`, and on others `ODBCSYSINI=/usr/local/etc`.

In particular, Micro Focus recommends that you set this environment variable if you want to use a SQL Fact Store with your Fact Bank systems, or Answer Bank statistics.

Install an IDOL Component as a Service on Windows

On Microsoft Windows operating systems, you can install any IDOL component as a Windows service. Installing a component as a Windows service makes it easy to start and stop the component, and you can configure a component to start automatically when you start Windows.

Use the following procedure to install Answer Server as a Windows service from a command line.

To install a component as a Windows service

1. Open a command prompt with administrative privileges (right-click the icon and select **Run as administrator**).
2. Navigate to the directory that contains the component that you want to install as a service.
3. Send the following command:

   Component`.exe -install`

   where *Component*`.exe` is the executable file of the component that you want to install as a service.

   The `-install` command has the following optional arguments:

`-start {[auto] | [manual] | [disable]}`	The startup mode for the component. **`Auto`** means that Windows services automatically starts the component. **`Manual`** means that you must start the service manually. **`Disable`** means that you cannot start the service. The default option is **`Auto`**.
`-username` *`UserName`*	The user name that the service runs under. By default, it uses a local system account.
`-password` *`Password`*	The password for the service user.
`-servicename` *`ServiceName`*	The name to use for the service. If your service name contains spaces, use quotation marks (") around the name. By default, it uses the executable name.
`-displayname` *`DisplayName`*	The name to display for the service in the Windows services manager. If your display name contains spaces, use quotation marks (") around the name. By default, it uses the service name.
`-depend` *`Dependency1`* `[,`*`Dependency2 ...`*`]`	A comma-separated list of the names of Windows services that Windows must start before the new service. For example, you might want to add the License Server as a dependency.

For example:

```
Component.exe -install -servicename ServiceName -displayname "Component Display
Name" -depend LicenseServer
```

After you have installed the service, you can start and stop the service from the Windows Services manager.

When you no longer require a service, you can uninstall it again.

To uninstall an IDOL Windows Service

1. Open a command prompt.
2. Navigate to the directory that contains the component service that you want to uninstall.
3. Send the following command:

   `Component``.exe -uninstall`

   where *`Component`*`.exe` is the executable file of the component service that you want to uninstall.

   If you did not use the default service name when you installed the component, you must also add the `-servicename` argument. For example:

   `Component``.exe -uninstall -servicename` *`ServiceName`*

Install an IDOL Component as a Service on Linux

On Linux operating systems, you can configure a component as a service to allow you to easily start and stop it. You can also configure the service to run when the machine boots. The following

procedures describe how to install Answer Server as a service on Linux.

> **NOTE:**
> To use these procedures, you must have `root` permissions.

> **NOTE:**
> When you install Answer Server on Linux, the installer prompts you to supply a user name to use to run the server. The installer populates the init scripts, but it does not create the user in your system (the user must already exist).

The procedure that you must use depends on the operating system and boot system type.

- For Linux operating system versions that use systemd (including CentOS 7, and Ubuntu version 15.04 and later), see Install a Component as a Service for a systemd Boot System, below.
- For Linux operating system versions that use System V, see Install a Component as a Service for a System V Boot System, on the next page.

Install a Component as a Service for a systemd Boot System

> **NOTE:**
> If your setup has an externally mounted drive that Answer Server uses, you might need to modify the init script. The installed init script contains examples for an NFS mount requirement.

To install an IDOL component as a service

1. Run the appropriate command for your Linux operating system environment to copy the init scripts to your `init.d` directory.
   - Red Hat Enterprise Linux (and CentOS)

     ```
     cp IDOLInstallDir/scripts/init/systemd/componentname
     /etc/systemd/system/componentname.service
     ```

   - Debian (including Ubuntu):

     ```
     cp IDOLInstallDir/scripts/init/systemd/componentname
     /lib/systemd/system/componentname.service
     ```

   where *componentname* is the name of the init script that you want to use, which is the name of the component executable (without the file extension).

   For other Linux environments, refer to the operating system documentation.
2. Run the following commands to set the appropriate access, owner, and group permissions for the component:
   - Red Hat Enterprise Linux (and CentOS)

     ```
     chmod 755 /etc/systemd/system/componentname
     chown root /etc/systemd/system/componentname
     chgrp root /etc/systemd/system/componentname
     ```

   - Debian (including Ubuntu):

```
chmod 755 /lib/systemd/system/componentname
chown root /lib/systemd/system/componentname
chgrp root /lib/systemd/system/componentname
```

where *componentname* is the name of the component executable that you want to run (without the file extension).

For other Linux environments, refer to the operating system documentation.

3. (Optional) If you want to start the component when the machine boots, run the following command:

```
systemctl enable componentname
```

Install a Component as a Service for a System V Boot System

To install an IDOL component as a service

1. Run the following command to copy the init scripts to your `init.d` directory.

```
cp IDOLInstallDir/scripts/init/systemv/componentname /etc/init.d/
```

where *componentname* is the name of the init script that you want to use, which is the name of the component executable (without the file extension).

2. Run the following commands to set the appropriate access, owner, and group permissions for the component:

```
chmod 755 /etc/init.d/componentname
chown root /etc/init.d/componentname
chgrp root /etc/init.d/componentname
```

3. (Optional) If you want to start the component when the machine boots, run the appropriate command for your Linux operating system environment:
   - Red Hat Enterprise Linux (and CentOS):

     ```
     chkconfig --add componentname
     chkconfig componentname on
     ```

   - Debian (including Ubuntu):

     ```
     update-rc.d componentname defaults
     ```

   For other Linux environments, refer to the operating system documentation.

Licenses

To use IDOL solutions, you must have a running License Server, and a valid license key file for the products that you want to use. Contact Micro Focus Big Data Support to request a license file for your installation.

License Server controls the IDOL licenses, and assigns them to running components. License Server must run on a machine with a static, known IP address, MAC address, or host name. The license key file is tied to the IP address and ACI port of your License Server and cannot be transferred between

machines. For more information about installing License Server and managing licenses, see the *License Server Administration Guide*.

When you start Answer Server, it requests a license from the configured License Server. You must configure the host and port of your License Server in the Answer Server configuration file.

You can revoke the license from a product at any time, for example, if you want to change the client IP address or reallocate the license.

CAUTION:
Taking any of the following actions causes the licensed module to become inoperable.

You **must not**:

- Change the IP address of the machine on which a licensed module runs (if you use an IP address to lock your license).
- Change the service port of a module without first revoking the license.
- Replace the network card of a client without first revoking the license.
- Remove the contents of the `license` and `uid` directories.

All modules produce a `license.log` and a `service.log` file. If a product fails to start, check the contents of these files for common license errors. See Troubleshoot License Errors, on page 22.

Display License Information

You can verify which modules you have licensed either by using the IDOL Admin interface, or by sending the `LicenseInfo` action from a web browser.

To display license information in IDOL Admin

- In the **Control** menu of the IDOL Admin interface for your License Server, click **Licenses**.

  The **Summary** tab displays summary information for each licensed component, including:
  - The component name.
  - The number of seats that the component is using.
  - The total number of available seats for the component.
  - (Content component only) The number of documents that are currently used across all instances of the component.
  - (Content component only) The maximum number of documents that you can have across all instances of the component.

  The **Seats** tab displays details of individual licensed seats, and allows you to revoke licenses.

To display license information by sending the LicenseInfo action

- Send the following action from a web browser to the running License Server.

`http://LicenseServerHost:Port/action=LicenseInfo`

where:

LicenseServerHost is the IP address of the machine where License Server resides.

Port	is the ACI port of License Server (specified by the `Port` parameter in the `[Server]` section of the License Server configuration file).

In response, License Server returns the requested license information. This example describes a license to run four instances of IDOL Server.

```
<?xml version="1.0" encoding="UTF-8" ?>
<autnresponse xmlns:autn="http://schemas.autonomy.com/aci/">
  <action>LICENSEINFO</action>
  <response>SUCCESS</response>
  <responsedata>
    <LicenseDiSH>
      <LICENSEINFO>
        <autn:Product>
          <autn:ProductType>IDOLSERVER</autn:ProductType>
          <autn:TotalSeats>4</autn:TotalSeats>
          <autn:SeatsInUse>0</autn:SeatsInUse>
        </autn:Product>
      </LICENSEINFO>
    </LicenseDiSH>
  </responsedata>
</autnresponse>
```

Configure the License Server Host and Port

Answer Server is licensed through License Server. In the Answer Server configuration file, specify the information required to connect to the License Server.

To specify the license server host and port

1. Open your configuration file in a text editor.
2. In the `[License]` section, modify the following parameters to point to your License Server.

`LicenseServerHost`	The host name or IP address of your License Server.
`LicenseServerACIPort`	The ACI port of your License Server.

   For example:

   ```
   [License]
   LicenseServerHost=licenses
   LicenseServerACIPort=20000
   ```

3. Save and close the configuration file.

Revoke a Client License

After you set up licensing, you can revoke licenses at any time, for example, if you want to change the client configuration or reallocate the license. The following procedure revokes the license from a component.

To revoke a license

1. Stop the IDOL solution that uses the license.
2. At the command prompt, run the following command:

   InstallDir/*ExecutableName*[.exe] -revokelicense -configfile *cfgFilename*

This command returns the license to the License Server.

You can send the `LicenseInfo` action from a web browser to the running License Server to check for free licenses. In this sample output from the action, one IDOL Server license is available for allocation to a client.

```
<autn:Product>
   <autn:ProductType>IDOLSERVER</autn:ProductType>
   <autn:Client>
      <autn:IP>192.123.51.23</autn:IP>
      <autn:ServicePort>1823</autn:ServicePort>
      <autn:IssueDate>1063192283</autn:IssueDate>
      <autn:IssueDateText>10/09/2003 12:11:23</autn:IssueDateText>
   </autn:Client>
      <autn:TotalSeats>2</autn:TotalSeats>
      <autn:SeatsInUse>1</autn:SeatsInUse>
</autn:Product>
```

Troubleshoot License Errors

The table contains explanations for typical licensing-related error messages.

License-related error messages

Error message	Explanation
Error: Failed to update license from the license server. Your license cache details do not match the current service configuration. Shutting the service down.	The configuration of the service has been altered. Verify that the service port and IP address have not changed since the service started.
Error: License for *ProductName* is invalid. Exiting.	The license returned from the License Server is invalid. Ensure that the license has not expired.
Error: Failed to connect to license server using cached licensed details.	Cannot communicate with the License Server. The product still runs for a limited period; however, you should verify whether your License Server is still available.
Error: Failed to connect to license server. Error code is SERVICE: *ErrorCode*	Failed to retrieve a license from the License Server or from the backup cache. Ensure that your License Server can be contacted.
Error: Failed to decrypt license keys. Please contact Autonomy support. Error code is SERVICE:*ErrorCode*	Provide Micro Focus Big Data Support with the exact error message and your license file.

License-related error messages, continued

Error message	Explanation
Error: Failed to update the license from the license server. Shutting down	Failed to retrieve a license from the License Server or from the backup cache. Ensure that your License Server can be contacted.
Error: Your license keys are invalid. Please contact Autonomy support. Error code is SERVICE:*ErrorCode*	Your license keys appear to be out of sync. Provide Micro Focus Big Data Support with the exact error message and your license file.
Failed to revoke license: No license to revoke from server.	The License Server cannot find a license to revoke.
Failed to revoke license from server *LicenseServer Host*:*LicenseServerPort*. Error code is *ErrorCode*	Failed to revoke a license from the License Server. Provide Micro Focus Big Data Support with the exact error message.
Failed to revoke license from server. An instance of this application is already running. Please stop the other instance first.	You cannot revoke a license from a running service. Stop the service and try again.
Failed to revoke license. Error code is SERVICE:*ErrorCode*	Failed to revoke a license from the License Server. Provide Micro Focus Big Data Support with the exact error message.
Your license keys are invalid. Please contact Autonomy Support. Error code is ACISERVER:*ErrorCode*	Failed to retrieve a license from the License Server. Provide Micro Focus Big Data Support with the exact error message and your license file.
Your product ID does not match the generated ID.	Your installation appears to be out of sync. Forcibly revoke the license from the License Server and rename the `license` and `uid` directories.
Your product ID does not match this configuration.	The service port for the module or the IP address for the machine appears to have changed. Check your configuration file.

Chapter 3: Configure Answer Server

The following sections describe how to configure Answer Server and the subcomponents that you need to use to run your systems.

- General Configuration 25
- Configure Answer Server Systems 26
  - Configure an Answer Bank System 26
  - Configure a Fact Bank System 26
  - Configure a Passage Extractor System 27
  - Configure a Conversation System 27
- Language Configuration 27
  - Answer Bank Language Configuration 28
  - Fact Bank and Passage Extractor Language Configuration 28
- Configure Client Authorization 29
- Configure SSL Communication Between Components 30
- Customize Logging 31
- Validate the Configuration File 32

General Configuration

The `[Server]` section of the Answer Server configuration file contains general settings that affect the server. In most cases, the only parameter that you might need to modify is the `Port` parameter, which controls the port that Answer Server listens on. This port must not be used by any other service on the host machine.

For example:

```
[Server]
Port=12000
```

Similarly, the `[Service]` section controls the behavior on the service port. You must make sure that the `ServicePort` parameter uses a port that is not used by any other service.

For example:

```
[Service]
ServicePort=12002
```

You must also configure the License Server for your system. For more information, see Configure the License Server Host and Port, on page 21.

Configure Answer Server Systems

To set up your answer server, you must configure one or more systems to use to retrieve questions. The `[Systems]` configuration section contains a list of systems that you want to configure.

The order in which you specify the systems is also the default order in which Answer Server requests answers from the systems. You can override this ordering for an individual action (see Ask Questions in Answer Server, on page 87).

```
[Systems]
0=MyFactBank
1=MyAnswerBank
```

For each of these systems, you then create a configuration section with the same name, which contains the settings for that system.

> **NOTE:**
> System names are case-sensitive.

Configure an Answer Bank System

For an Answer Bank system, you must set `Type` to **`answerbank`**. You must also configure the host and port of the Agentstore component that you are using as the Answer Bank.

```
[MyAnswerBank]
Type=answerbank
IDOLHost=localhost
IDOLACIPort=6000
```

For more information, see Set Up an Answer Bank System, on page 39.

Configure a Fact Bank System

For a Fact Bank system, you must set `Type` to **`factbank`**. You must also configure the question parser grammars, the fact store, and the location of the coding files.

```
[MyFactBank]
Type=factbank
// Question Parser
EductionQuestionGrammars=FactBankEductionGrammar.ecr
EductionTimeGrammars=datetime_processing.ecr
EductionLuaScript=lua/FactBankEduction.lua
TimeLuaScript=lua/DateTime.lua
// Fact Store
BackendType=sqldb
ConnectionString=Driver=PostgreSQL ANSI(x64); Server=sql-host.mycompany.com;
Port=5432; Database=factstoredb; Uid=postgres;password=password;// Coding Files
CodingsPath=./codings
CodingsDatPath=./codings
```

For more information, see Set Up a Fact Bank System, on page 57.

Configure a Passage Extractor System

For a Passage Extractor system, you must set `Type` to **`passageextractor`**. You must also configure the host and port of the IDOL Content component data store, as well as the Eduction grammars and Agentstore components to use for entity extraction. You can also optionally define the locations of the classifier file and label file to allow you to save your training classifiers.

```
[MyPassageExtractor]
Type=PassageExtractor
// Data store IDOL
IdolHost=localhost
IdolAciport=6002
// Entity Agentstore
AgentStoreHost=localhost
AgentStoreAciport=5002
// Eduction
EductionGrammars=pe_grammar.ecr,NUM.ecr,HUM.ecr,date_eng.ecr,money_eng.ecr
// Classifier Files
ClassifierFile=en_svm.dat
LabelFile=en_labels.dat
```

Configure a Conversation System

For a Conversation system, you must set `Type` to **`conversation`**. You must also configure the location of a task configuration file, which defines the conversation task in more detail. You can also optionally define the location of an Agentstore component to use to store conversation trigger agents, and session expiration for the conversation sessions.

```
[MyConversation]
Type=Conversation
TaskConfigurationFile=C:\AnswerServer\Conversation\tasks.json
// Trigger Agentstore
AgentStoreHost=localhost
AgentStoreAciport=5002
// Session Expiration
SessionExpirationIdleTime=600
SessionExpirationInterval=60
```

Language Configuration

The Answer Server functionality uses language-dependent information to parse and classify questions and match them to answers.

Answer Bank Language Configuration

In the answer bank systems, the language-dependent processing is managed by the answer bank Agentstore component. The Agentstore component stores the questions, answers, and processes the `Ask` actions as queries.

You can configure languages in the Agentstore in the same way as in the IDOL Content component. For more information, refer to the *IDOL Server Reference* and the *IDOL Server Administration Guide*.

Fact Bank and Passage Extractor Language Configuration

In fact bank and passage extractor systems, Answer Server can use stemming and stop lists to improve the question parsing and answer matching.

- *Stemming* is the process of reducing related words, such as plurals and verb forms, to a common linguistic root. For example, in English, *helping*, *helped*, and *helps* all derive from the common root *help*.

  Stemming rules are language-dependent. To get the best possible results, you must specify the language that you use in your questions.
- A *stop list* is a list of very common words, which usually add very little meaning to phrases. For example, in English, *the* and *and* can often be ignored without losing the sense of a sentence. IDOL uses stop lists to optimize matching.

  In a fact bank system, Answer Server uses the stop list to match fact bank codes more broadly. Answer Server attempts to form pseudonym values in the code maps by taking the existing code phrases and removing stop words from the beginning and end of the phrase. Similarly, when Answer Server attempts to match a string to codes, it matches the full phrase, and the phrase with the stop words removed from the beginning and end of the phrase. Answer Server does not attempt to remove stop words from the middle of phrases.

  In a passage extractor system, Answer Server removes stop words from the classified questions to attempt to find the best match in the data IDOL Content component. Answer Server does not use the stop list for question classification, which often depends on common words, such as question words.

By default, Answer Server uses English stemming rules, and does not use a stop list. If you use the default fact bank and passage extractor grammar files, this is usually appropriate, although you might want to add an English stop list, by setting the `StopList` configuration parameter.

To use fact bank and passage extractor with different languages, you need a version of the grammar files in the appropriate language. These grammar files are available in English, French, German, Italian, and Spanish. If you are interested in using fact bank and passage extractor in other languages, contact your Micro Focus account manager.

If you are using fact bank and passage extractor in a language other than English, you can change the stemming language by modifying the `Language` configuration parameter. You can also add a stop list by setting the `StopList` configuration parameter.

You can configure the location where you store your language files (such as stop lists), by using the `LanguageDirectory` configuration parameter.

You can define language configuration in the `[Server]` section, to apply to all your systems. You can also set the language configuration parameters in your individual system configuration sections. If you set the parameters in both sections, the system configuration takes precedence.

For more information about these parameters, refer to the *Answer Server Reference*.

Configure Client Authorization

You can configure Answer Server to authorize different operations for different connections.

Authorization roles define a set of operations for a set of users. You define the operations by using the `StandardRoles` configuration parameter, or by explicitly defining a list of allowed actions in the `Actions` and `ServiceActions` parameters. You define the authorized users by using a client IP address, SSL identities, and GSS principals, depending on your security and system configuration.

For more information about the available parameters, see the *Answer Server Reference*.

To configure authorization roles

1. Open your configuration file in a text editor.
2. Find the `[AuthorizationRoles]` section, or create one if it does not exist.
3. In the `[AuthorizationRoles]` section, list the user authorization roles that you want to create. For example:

   ```
   [AuthorizationRoles]
   0=AdminRole
   1=UserRole
   ```

4. Create a section for each authorization role that you listed. The section name must match the name that you set in the `[AuthorizationRoles]` list. For example:

   ```
   [AdminRole]
   ```

5. In the section for each role, define the operations that you want the role to be able to perform. You can set `StandardRoles` to a list of appropriate values, or specify an explicit list of allowed actions by using `Actions`, and `ServiceActions`. For example:

   ```
   [AdminRole]
   StandardRoles=Admin,ServiceControl,ServiceStatus

   [UserRole]
   Actions=GetVersion
   ServiceActions=GetStatus
   ```

   > **NOTE:**
   > The standard roles do not overlap. If you want a particular role to be able to perform all actions, you must include all the standard roles, or ensure that the clients, SSL identities, and so on, are assigned to all relevant roles.

6. In the section for each role, define the access permissions for the role, by setting `Clients`, `SSLIdentities`, and `GSSPrincipals`, as appropriate. If an incoming connection matches one of the allowed clients, principals, or SSL identities, the user has permission to perform the operations

allowed by the role. For example:

```
[AdminRole]
StandardRoles=Admin,ServiceControl,ServiceStatus
Clients=localhost
SSLIdentities=admin.example.com
```

7. Save and close the configuration file.
8. Restart Answer Server for your changes to take effect.

Configure SSL Communication Between Components

You can configure Answer Server to use SSL to connect to each of its back end components, such as the IDOL Content and Agentstore components that you use in your systems.

For each child component, there is a *Component*`Host` and an *Component*`ACIPort` configuration parameter in the system configuration, which specifies how to connect to the component. You can also add the *Component*`SSLConfig` parameter. For example, for the IDOL Content component, you use `IDOLHost`, `IDOLACIPort`, `IDOLSSLConfig`.

You set the *Component*`SSLConfig` parameter to the name of the configuration section where you define the SSL settings for connection to that component. By convention, this section has the name `SSLOption`*N*.

> **NOTE:**
> For the Agentstore and Content components, Answer Server uses the same SSL configuration for ACI and indexing requests.

For example:

```
[MyPassageExtractor]
Type=passageextractor
// Data Content
IDOLHost=localhost
IDOLACIPort=10050
IDOLSSLConfig=SSLOption1
// Entity Agentstore
AgentstoreHost=localhost
AgentstoreACIPort=10060
IDOLSSLConfig=SSLOption0
...

[SSLOption0]
SSLMethod=TLSV1.2
SSLCertificate=host1.crt
SSLPrivateKey=host1.key
SSLCACertificate=trusted.crt

[SSLOptions1]
SSLMethod=TLSV1.2
SSLCertificate=host2.crt
```

```
SSLPrivateKey=9s7BxMjD2d3M3t7awt/J8A
SSLCACertificate=trusted.crt
```

For more information about the SSL configuration options, refer to the *Answer Server Reference*.

Customize Logging

You can customize logging by setting up your own *log streams*. Each log stream creates a separate log file in which specific log message types (for example, action, index, application, or import) are logged.

To set up log streams

1. Open the Answer Server configuration file in a text editor.
2. Find the `[Logging]` section. If the configuration file does not contain a `[Logging]` section, add one.
3. In the `[Logging]` section, create a list of the log streams that you want to set up, in the format *`N=LogStreamName`*. List the log streams in consecutive order, starting from 0 (zero). For example:

   ```
   [Logging]
   LogLevel=FULL
   LogDirectory=logs
   0=ApplicationLogStream
   1=ActionLogStream
   ```

   You can also use the `[Logging]` section to configure any default values for logging configuration parameters, such as `LogLevel`. For more information, see the *Answer Server Reference*.
4. Create a new section for each of the log streams. Each section must have the same name as the log stream. For example:

   ```
   [ApplicationLogStream]
   [ActionLogStream]
   ```

5. Specify the settings for each log stream in the appropriate section. You can specify the type of logging to perform (for example, full logging), whether to display log messages on the console, the maximum size of log files, and so on. For example:

   ```
   [ApplicationLogStream]
   LogTypeCSVs=application
   LogFile=application.log
   LogHistorySize=50
   LogTime=True
   LogEcho=False
   LogMaxSizeKBs=1024

   [ActionLogStream]
   LogTypeCSVs=action
   LogFile=logs/action.log
   LogHistorySize=50
   LogTime=True
   ```

```
LogEcho=False
LogMaxSizeKBs=1024
```

6. Save and close the configuration file. Restart the service for your changes to take effect.

Validate the Configuration File

You can use the `ValidateConfig` service action to check for errors in the configuration file.

> **NOTE:**
> For the `ValidateConfig` action to validate a configuration section, Answer Server must have previously read that configuration. In some cases, the configuration might be read when a task is run, rather than when the component starts up. In these cases, `ValidateConfig` reports any unread sections of the configuration file as unused.

To validate the configuration file

- Send the following action to Answer Server:

  `http://`*`Host`*`:`*`ServicePort`*`/action=ValidateConfig`

  where:

`Host`	is the host name or IP address of the machine where Answer Server is installed.
`ServicePort`	is the service port, as specified in the `[Service]` section of the configuration file.

Chapter 4: Run Answer Server

This section describes how to start and stop Answer Server, and to send actions.

- Start Answer Server 33
- Stop Answer Server 33
- Verify that Answer Server is Running 34
  - GetStatus 34
  - GetLicenseInfo 34
- Send Actions to Answer Server 35

Start Answer Server

> **NOTE:**
> Your License Server must be running before you start Answer Server.

To start Answer Server

- Start Answer Server from the command line using the following command:

  ```
  ServerName.exe -configfile configname.cfg
  ```

  where *ServerName* is the name of the server executable file, and the optional `-configfile` argument specifies the path of a configuration file that you want to use.
- On Windows, if you have installed Answer Server as a service, start the service from the Windows Services dialog box.
- On UNIX, if you have installed Answer Server as a service, use one of the following commands:
  - On machines that use systemd:

    ```
    systemctl start ServerName
    ```
  - On machines that use system V:

    ```
    service ServerName start
    ```

> **TIP:**
> On both Windows and UNIX platforms, you can configure services to start automatically when you start the machine.

Stop Answer Server

You can stop Answer Server by using one of the following procedures.

To stop Answer Server

- Send the `Stop` service action to the service port.

  `http://host:ServicePort/action=Stop`

  where:

host	is the host name or IP address of the machine where Answer Server is installed.
ServicePort	is the Answer Server service port (specified in the `[Service]` section of the configuration file).

- On Windows platforms, if Answer Server is running as a service, stop Answer Server from the Windows Services dialog box.
- On UNIX platforms, if Answer Server is running as a service, use one of the following commands:
  - On machines that use systemd:

    `systemctl stop ServerName`
  - On machines that use system V:

    `service ServerName stop`

Verify that Answer Server is Running

After starting Answer Server, you can run the following actions to verify that Answer Server is running.

- GetStatus
- GetLicenseInfo

GetStatus

You can use the `GetStatus` service action to verify the Answer Server is running. For example:

`http://Host:ServicePort/action=GetStatus`

> **NOTE:**
> You can send the `GetStatus` action to the ACI port instead of the service port. The `GetStatus` ACI action returns information about the Answer Server setup.

GetLicenseInfo

You can send a `GetLicenseInfo` action to Answer Server to return information about your license. This action checks whether your license is valid and returns the operations that your license includes.

Send the `GetLicenseInfo` action to the Answer Server ACI port. For example:

`http://Host:ACIport/action=GetLicenseInfo`

The following result indicates that your license is valid.

```
<autn:license>
       <autn:validlicense>true</autn:validlicense>
</autn:license>
```

As an alternative to submitting the `GetLicenseInfo` action, you can view information about your license, and about licensed and unlicensed actions, on the **License** tab in the Status section of IDOL Admin.

Send Actions to Answer Server

Answer Server actions are HTTP requests, which you can send, for example, from your web browser. The general syntax of these actions is:

`http://`*`host`*`:`*`port`*`/action=`*`action`*`&`*`parameters`*

where:

`host`	is the IP address or name of the machine where Answer Server is installed.
`port`	is the Answer Server ACI port. The ACI port is specified by the `Port` parameter in the `[Server]` section of the Answer Server configuration file. For more information about the `Port` parameter, see the *Answer Server Reference*.
`action`	is the name of the action you want to run.
`parameters`	are the required and optional parameters for the action.

> **NOTE:**
> Separate individual parameters with an ampersand (&). Separate parameter names from values with an equals sign (=). You must percent-encode all parameter values.

For more information about actions, see the *Answer Server Reference*.

Part II: Configure Answer Server Systems

This section describes how to configure and use the different types of systems in Answer Server.

- Set Up an Answer Bank System, on page 39
- Set Up a Fact Bank System, on page 57
- Set Up a Passage Extractor System, on page 75
- Ask Questions in Answer Server, on page 87
- Set Up a Conversation System, on page 89
- Hold Conversations in Answer Server, on page 125

Chapter 5: Set Up an Answer Bank System

This section describes how to set up an Answer Bank system, and add questions, question equivalence classes, and answers.

- Configure the Answer Bank System 39
- Configure the Answer Bank Agentstore 40
- Manage an Answer Bank 40
- Store Statistics for Your Answer Bank 53
- Retrieve the Information Stored in an Answer Bank 54

Configure the Answer Bank System

The Answer Server configuration file contains information about the subcomponents in your Answer Bank systems.

For any Answer Bank system, you must configure the host and port details of the Answer Bank Agentstore, which is a specially configured IDOL Content component that stores your questions, answers, and question equivalence classes.

The following procedure describes how to configure the Answer Bank system in Answer Server.

There are also several optional parameters in the Answer Bank system, to allow you to modify the paths that Answer Bank uses for failed and queued actions, and how often it updates the Answer Bank usage statistics.

For more details about the configuration parameters for the Answer Bank system, refer to the *Answer Server Reference*.

To configure the Answer Bank System

1. Open the Answer Server configuration file in a text editor.
2. Find the `[Systems]` section, or create one if it does not exist. This section contains a list of systems, which refer to the associated configuration sections for each system.
3. After any existing systems, add an entry for your new Answer Bank system. For example:

   ```
   [Systems]
   0=MyFactBank
   1=MyAnswerBank
   ```

4. Create a configuration section for your Answer Bank system, with the name that you specified. For example, `[MyAnswerBank]`.
5. Set `Type` to **`AnswerBank`**.
6. Set `IDOLHost` and `IDOLACIPort` to the host name and ACI Port of the IDOL Agentstore component that contains the questions and answers.
7. Save and close the configuration file.
8. Restart Answer Server for your changes to take effect.

For example:

```
[MyAnswerBank]
Type=answerbank
IDOLHost=localhost
IDOLACIPort=6000
```

Configure the Answer Bank Agentstore

An Answer Bank system requires an IDOL Agentstore component, which is a specialized configuration of the IDOL Content component.

The Answer Server package includes a predefined configuration file for the Answer Bank Agentstore component, which includes the relevant field and server configuration for Answer Server. You must download and install the IDOL Content component separately, and then update the configuration file with the Answer Bank Agentstore-specific configurations.

To configure the Agentstore component for your Answer Bank

1. In your Answer Server installation directory, copy the Answer Bank Agentstore configuration file (`agentstore.cfg`).
2. Open your IDOL Content component installation directory.
3. Paste the Answer Server Agentstore configuration file. Overwrite the installed configuration file (you might want to make a copy of it first).

   > **NOTE:**
   > The configuration file must have the same name as the executable file.

4. Open the configuration file in a text editor.
5. Update the `[License]` section with the host and port information for your License Server. For more information, see Configure the License Server Host and Port, on page 21.
6. Find the `[Server]` section `Port` parameter. Check that the specified port is available on the host machine, or change it to an available port.
7. Find the `[Service]` section `ServicePort` parameter. Check that the specified port is available on the host machine, or change it to an available port.
8. Save and close the configuration file.

Manage an Answer Bank

You manage the questions and answers in your Answer Bank by using the `ManageResource` action. This action allows you to upload a JSON object that contains the changes that you want to make in the Answer Bank.

You can also use the IDOL Data Admin user interface to manage the questions and answers in your Answer Bank. For more information, refer to the *IDOL Data Admin User Guide*.

Find the JSON Schema for Your Update

The schemas for the `ManageResources` action are stored in Answer Server. You can retrieve them by using the `GetResources` action. You can restrict this action to a particular system in your configuration file by specifying the `SystemName` parameter.

For example:

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=Schema
```

> **NOTE:**
> System names are case sensitive. The value that you specify in the `SystemName` parameter must match the name of the system in the configuration file.

Add a Question

You can use the add question operation to keep track of questions that your users ask, and to make rules based on real questions that hopefully match future questions. For example, you might have your user interface set up to add every question that does not have an answer to the Answer Bank, to build up a list of frequently asked questions, which you can add answers for.

To add a question, you use a `ManageResources` action in a POST request method, with the update provided in the `Data` parameter as a JSON object.

The following simple example adds a question to the `AnswerBank` system.

```
Action=ManageResources&SystemName=AnswerBank
data={
   "operation": "add",
   "type":"question",
   "questions":[
      {"text":"Where do I sign up for the monthly newsletter?"}
   ]
}
```

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, above.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

The action returns a `question_id`, which you can use to create and update question equivalence classes and add answers.

Find the Likelihood of Existing Answers

When you add new questions to your Answer Bank Agentstore, you can find out whether there are any likely answers in your existing question equivalence classes.

The answer likelihood score field for a particular question stores the likelihood that there is an existing answer. Answer Server uses the question text to query your question equivalence classes, and uses the relevance score of each question equivalence class that returns in the query to calculate the likelihood score.

You can use this score to sort questions by likelihood in the `GetResources` action. See GetResources, on page 55.

Periodically, Answer Server runs a background process to calculate the likelihood scores for all the questions in the Answer Bank Agentstore that do not have an answer, and updates the field for those questions.

By default, this process runs every 600 seconds (ten minutes). You can use the `UpdateLikelihoodInterval` configuration parameter in your Answer Bank system configuration to change how frequently to update the likelihood score field.

You can update the field more frequently if you need up-to-date information to sort by likelihood. However, for performance reasons Micro Focus recommends that you do not update the likelihood score field too frequently, because it might result in a large number of indexing operations in the Answer Bank Agentstore component.

You can set `UpdateLikelihoodInterval` to a higher value if your question equivalence classes do not change very often, or if do not intend to use the likelihood scores very often.

For more information about `UpdateLikelihoodInterval` and the `GetResources` sort options, refer to the *Answer Server Reference*.

Find the Likely Answers to a Question

You can use the `GetResources` action to filter the list of question equivalence classes to those that are likely to provide an answer for a particular question or set of questions, by using the `likely_answer_for` filter. This filter matches question equivalence classes where the answer is most relevant to the specified questions.

For example:

```
action=GetResources&type=question_equivalance_class&filter={ "likely_answer_for": [
{"ids":[ "2373534828452857425" ], "resource_type":"question"} ] }
```

For more information, refer to the *Answer Server Reference*.

Question Equivalence Rules

The question equivalence rule provides a general expression to match several equivalent questions. The rule forms part of a question equivalence class. When you ask a question, Answer Server matches the question text against the rules in your question equivalence classes.

In general, you use text with a Boolean or proximity expression to match questions.

For example, the questions *Why is the sky blue?*, *What causes the sky to appear blue?*, and *What gives the sky the color blue?* are all equivalent. You might use the rule `sky NEAR blue` to match all these questions.

You can expand the rules to more complicated expressions to match more complicated sets of equivalent questions. The expression can use the same Boolean and proximity syntax as the

IDOL Content component query text, along with bracketed expressions to specify priority. For more information, refer to the *IDOL Server Reference*.

> **NOTE:**
> You cannot use Wildcard expressions in your question equivalence rules unless there is also a non-Wildcard term required by the rule. For example, you cannot use the rule `sk*`, because it contains only a Wildcard term. However, you can use `sk* AND blue`, because blue is also required. Answer Server returns an error if you try to use an invalid rule.
>
> In general, Micro Focus recommends that you do not use Wildcard expressions in your rules.

In all these cases, you add the rule to the question equivalence class by using the `ManageResources` action. The `add` and `update` operations for question equivalence classes accept a `rule` property, which accepts the expression as a string value. For example:

```
"rule":"sky NEAR blue",
```

For these rules, you can use the `GetResources` action to suggest rules. See Generate a Question Equivalence Rule, on the next page. You can also use the `TestRule` action to test whether a rule matches all the questions you want. See Test Your Question Equivalence Rule, on page 45

For more information about adding a rule to a question equivalence class, see Create a Question Equivalence Class and Add an Answer, on page 47 and Update a Question Equivalence Class, on page 48.

Special Rule Types

In certain special cases, you might also want to use a `FieldText` expression. This option is most useful when you have a very short question that you want to match exactly. In this case, you can use a `FieldText` expression to prevent the rule from matching any longer question that contains your short question.

For example, the question *What is life?* is very short, and might easily match longer questions that are not equivalent, such as *What is Life of Pi about?*.

When you want to use a `FieldText` expression when you add or update a question equivalence class, you set the `rule` property as an object. This object has a required `text` property, and an optional `fieldtext` property, which set the appropriate text and `FieldText` rules.

The `Ask` action sends the question to your Answer Bank Agenstore in a `DRECONTENT` field. You can use your `FieldText` expression to match text in this field by using the IDOL Content component `FieldText` operators.

For example:

```
"rule": { "text": "What is Life?", "fieldtext": "MATCH{What is Life?}:DRECONTENT" }
```

This rule matches only the exact question *What is life?*

For more information about `FieldText` operators, refer to the *IDOL Server Reference*.

> **NOTE:**
> You must always specify the `text` property. That is, you cannot create a rule with only `FieldText`.

The more complicated rule types can be useful in certain circumstances. However, the `GetResources` rule suggestion option does not return any suggestions that contain `FieldText`. Similarly, you cannot test complex `FieldText` rules by using the `TestRule` action.

Generate a Question Equivalence Rule

You can use the `GetResources` action to suggest a question equivalence rule, based on a set of questions. You can use this to automatically generate an initial question equivalence rule, which you can modify to optimize the rule for your question equivalence class.

For example:

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=rule_
suggestion&IDs=9706856188043740111,8129920660480699726,3067998369792637739
```

This example generates a rule that matches the questions with the IDs `9706856188043740111`, `8129920660480699726`, and `3067998369792637739`.

You can also optionally add additional questions in a text filter, for example to specify the reference question for the question equivalence class that you want to create, or to include questions that do not currently exist in the Answer Bank index.

For example:

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=rule_
suggestion&IDs=9706856188043740111,8129920660480699726,3067998369792637739&Filter=%
7B%20%22rule_suggestion_
text%22%20%3A%20%5B%22How%20do%20I%A0get%20regular%20updates%20about%20MyCompany%3F
%22%5D%20%7D
```

The `Filter` parameter takes a percent-encoded JSON object, which contains the filters to apply. In this case, the unencoded JSON object is:

```
{ "rule_suggestion_text" : ["How do I get regular updates about MyCompany?"] }
```

> **TIP:**
> You can also use `GetResources` to find question equivalence classes that do not have an associated rule, by setting the `Type` parameter to **question_equivalence_class**, and using the `Filter` parameter to search for the `incoming` state (which is for classes that do not have rules). For example:
>
> ```
> Action=GetResources&SystemName=MyAnswerbank&Type=question_equivalence_
> class&Filter=%7B%20%22state%22%3A%20%22incoming%22%20%7D
> ```
>
> This corresponds to the following filter:
>
> ```
> { "state" : "incoming" }
> ```

You can modify the question equivalence rule as required, and add it to the question equivalence class by using the `ManageResources` action. See Update a Question Equivalence Class, on page 48.

You can also test that the question equivalence rule matches all the rules that you want to add to the question class by using the `GetResources` action. See Test Your Question Equivalence Rule, on the next page.

Test Your Question Equivalence Rule

When you create a question equivalence rule, you can check that it matches all the questions that you want to include in the question equivalence class, and that it does not match questions that belong to other question equivalence classes. The `GetResources` action has several filters that you can use to test your rules.

You can also use the `TestRule` action to test whether a particular set of questions matches a rule that you specify. In this case, you do not need to have indexed the questions or the question equivalence rule.

> **NOTE:**
> The `TestRule` action can test only rules that use simple text (with any Boolean and proximity expressions). You cannot test `FieldText` rules in this way. For more information about `FieldText` rules, see Special Rule Types, on page 43.

Find Questions that Do Not Match the Rule

You can add the question equivalence rule text as a text filter in `GetResources`, to find questions that match or do not match the rule. You can also specify a list of IDs for questions that you want to match the rule in the `IDs` parameter.

> **TIP:**
> You can retrieve a list of question IDs and the rule for a question equivalence class by sending a `GetResources` action with `Type` set to **`question_equivalence_class`**.

To test whether the rule matches the questions in the question equivalence class, set the text filter to `NOT(`*`RuleText`*`)`, and set `IDs` to a list of IDs of the questions in that class.

For example:

```
http://localhost:12000?Action=GetResources&Type=question&IDs=9706856188043740111,81
29920660480699726,306799836979263773 9&Filter=%7B%20%22text%22%20%3A%20%22NOT
(updates%20AND%C2%A0MyCompany)%22%20%7D
```

The `Filter` parameter takes a percent-encoded JSON object, which contains the filters to apply. In this case, the unencoded JSON object is:

```
{ "text" : "NOT(updates AND MyCompany)" }
```

This action returns any questions in the list of IDs that do not match the question equivalence rule `updates AND MyCompany`.

Find Questions in Other Classes that Match the Rule

After you create the question equivalence class and add the rule, you can also use `GetResources` to find out if your rule matches any questions that do not belong to the class, by using the `not_associated_with` filter.

For example, for a `GetResources` action with `Type` set to **`question`**, the following filter object matches questions that match the rule `updates AND MyCompany`, and that do not belong to the question equivalence class with ID 1429393462892614629.

```
{
   "text" : "updates AND MyCompany",
   "not_associated_with": [
      {
         "ids": [
            "1429393462892614629"
         ],
         "type": "question_equivalence_class"
      }
   ]
}
```

This filter returns questions that match the rule that already belong to a different question equivalence class. This might indicate that your rule is not restrictive enough.

> **TIP:**
> This filter also returns any questions that match the rule and that do not belong to a question equivalence class. If you want to find only questions that belong to a different question equivalence class, you can add an additional `state` filter to find questions in the `answered` state.

Test Whether Questions Match a Specified Rule

You can use the `TestRule` action to test questions and rules that you have not added to the Answer Bank Agentstore. This action allows you to test rules and questions before you index them, to save indexing time, and reindexing time if you need to change the rule.

To use `TestRule`, you specify the questions in a JSON object in the `Questions` parameter, and the rule in the `Rule` parameter. You must also set `SystemName` to the name of the Answer Bank system.

For example:

```
action=TestRule&SystemName=MyAnswerBank&Questions={"text":["Why is the sky
blue?","What causes the sky to appear blue?","How do I return a defective
item?"],"ids":
["7660794084496396635","15927917885427259786","14042282250303108454"]}&Rule=sky AND
blue
```

This action tests whether the questions *Why is the sky blue?*, *What causes the sky to appear blue?*, and *How do I return a defective item?*, and the questions with IDs 7660794084496396635, 15927917885427259786, and 14042282250303108454 match the rule `sky AND blue`.

The action returns the rule, with a `matched` property that contains the list of matching questions, and the `not_matched` property that contains any questions in the request that do not match the rule.

```
{
   "rule": "sky AND blue",
   "matched": {
      "text": [
         "Why is the sky blue?",
         "What causes the sky to appear blue?"
      ],
```

```
      "id" [
         "15927917885427259786",
         "14042282250303108454"
      ]
   },
   "not_matched": {
      "text": [
         "How do I return a defective item?"
      ],
      "id": [
        "7660794084496396635"
      ]
   }
}
```

Create a Question Equivalence Class and Add an Answer

A question equivalence class is a group of equivalent questions. When you have multiple questions with different wording that request the same information, you can combine these questions in a question equivalence class. You also use the question equivalence class to add an answer.

The question equivalence class contains the following information:

- the reference question (a standard question that is the most representative question for the answer).
- the answer to the reference question.
- a list of real, sample questions that are equivalent to the reference question. These questions are stored by adding a question. See Add a Question, on page 41.
- a question equivalence rule that matches the equivalence question. When a user asks a question, Answer Server uses the question equivalence rule to find a relevant question equivalence class to match. See Question Equivalence Rules, on page 42.

To create question equivalence classes, you use a `ManageResources` action in a POST request method, with the information about the questions provided in the `Data` parameter as a JSON object.

> **NOTE:**
> When you add a question to a question equivalence class, Answer Server automatically updates the question state to `answered`. You cannot delete a question in this state, and you cannot manually move the question out of this state. You must remove the question from the question equivalence class. See Update a Question Equivalence Class, on the next page.

You must include the `question_id` values for the questions that you want to include in the question equivalence class. You can retrieve the `question_id` values by sending a `GetResources` action. For example:

```
http://localhost:12000?Action=GetResources&SystemName=Answerbank&Type=question
```

You can add multiple question equivalence classes in the same operation, where each question equivalence class is an object in an array.

The following example adds a single question equivalence class that combines two questions, and adds a reference question and an answer. You can also add optional metadata for the answer, such as author information.

```
Action=ManageResources&SystemName=AnswerBank
data={
   "operation":"add",
   "type":"question_equivalence_class",
   "question_equivalence_classes":[
      {
         "question_ids":[
            "9706856188043740111",
            "8129920660480699726"
         ],
         "rule":{"text":"how AND (regular NEAR2 updates) AND MyCompany"},
         "answer":{
            "text":"Send an email to subscribe@example.com, and we'll add you to
our monthly newsletter.",
            "metadata":[
               {"key":"author", "value":"Alex"},
               {"key":"modified_date", "value":"2017-05-05"}
            ]
         },
         "reference_question":"How do I get regular updates about MyCompany?"
      }
   ]
}
```

> **NOTE:**
> Metadata key names can contain alphanumeric characters (a-z, 0-9), period (.), underscore (_) and hyphen (-). They cannot start with a number. It is also best practice to use field names that conform to XML specifications.

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

You can also use the `GetResources` action to generate a rule for your question equivalence class. See Generate a Question Equivalence Rule, on page 44.

Update a Question Equivalence Class

You can update the following properties in your question equivalence classes:

- the list of question IDs that the question equivalence class contains.
- the answer to the reference question.
- the question equivalence rule.

- the text of the reference question. In general, Micro Focus recommends you use this update to correct spelling mistakes only. If you want to completely change the reference question, create a new question equivalence class with a new ID.

To update a question equivalence class, you use a `ManageResources` action in a POST request method, with the information about the questions provided in the `Data` parameter as a JSON object.

The `update` operation replaces existing content with the new version that you provide. If you omit a value, Answer Server keeps the existing value. If you want to delete the existing value, you can explicitly set an empty value or array.

For example, if you change the list of question IDs, but do not set the answer block in your JSON data, Answer Server updates the list of questions, but does not modify the answer.

The `amend` operation allows you to add questions to the question equivalence class. In this case, you provide only the questions that you want to add to the question equivalence class, rather than the complete list of question IDs. You can use this option if you might have multiple users adding equivalent questions at the same time, to ensure that all changes are reflected.

> **NOTE:**
> When you add questions to a question equivalence class, Answer Server automatically updates the question state to `answered`. If you remove all questions from the question equivalence class, it reverts the question state to `incoming`.

The following example updates the question equivalence class with ID 2012912839742797651 with a new list of `question_ids`. It does not update the stored answer for the question (if it exists).

```
Action=ManageResource&SystemName=AnswerBank
data={
   "operation":"update",
   "type":"question_equivalence_class",
   "question_equivalence_class":{
      "id": 2012912839742797651,
      "question_ids":[
         "9706856188043740111",
         "8129920660480699726",
         "3067998369792637739"
      ],
      "reference_question":"How do I get regular updates about MyCompany?",
      "rule": "updates AND MyCompany"
   }
}
```

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

You can also use the `GetResources` action to generate a rule for your question equivalence class. See Generate a Question Equivalence Rule, on page 44.

Update the Question State

The questions in your Answer Bank systems have a state, which can have one of the following values:

- `incoming`
- `answerable`
- `needs_answer`
- `answered`
- `rejected`

Answer Server makes some changes to the question state automatically. You an also modify the question state manually by using the `ManageResources` action.

Automatic Question State Updates

When you add a question to the Answer Bank, Answer Server automatically assigns it the state `incoming`.

When you add the question to a question equivalence class, the state automatically updates to `answered`.

Similarly, if you remove the question from the question equivalence class (or if you delete the question equivalence class), the state reverts to `incoming`. If you later undelete the question equivalence class that the question belongs to, the question returns to `answered` again.

Update the Question State Manually

You can modify the state of a question by using the `ManageResources` action, with the `update` operation, either as part of another update, or as a separate action.

> **NOTE:**
> You cannot manually change the state to or from `answered` by using a question update. To move the question to the `answered` state, you must add the question to a question equivalence class.

You must include the question IDs for the questions that you want to update. You can retrieve the question ID by sending a `GetResources` action. For example:

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=question
```

The following example updates the questions with IDs 9706856188043740111 and 8129920660480699726 to have the state `needs_answer`.

```
Action=ManageResources&SystemName=AnswerBank
data={
   "operation": "update",
   "type": "question",
   "question": {
      "question_ids": [
         "9706856188043740111",
```

```
         "8129920660480699726"
      ],
   "new_state": "needs_answer"
   }
}
```

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

You can use the question state to filter questions in the `GetResources` action. You can also use the `GetStats` action to return information about the questions in the Answer Bank system by state. For more information, see Retrieve the Information Stored in an Answer Bank, on page 54.

Delete a Question or Question Equivalence Class

You can use the `ManageResources` action to delete a question or a question equivalence class.

After you delete an item, there is a short period in which you can undelete it, before it is permanently deleted from the system. See Undelete a Question or Question Equivalence Class, on the next page.

To delete a question or question equivalence class, you use a `ManageResources` action in a POST request method, with the information about the item that you want to delete provided in the `Data` parameter as a JSON object.

You must include the ID of the item that you want to delete, and set the type of the operation to `question` or `question_equivalence_class`, as appropriate.

> **NOTE:**
> You cannot delete a question that has the state `answered` (that is, a question that belongs to a question equivalence class). You must remove it from the question equivalence class before you delete it. See Update the Question State, on the previous page and Update a Question Equivalence Class, on page 48.

You can retrieve the ID of the question or question equivalence class by sending a `GetResources` action. For example:

```
http://localhost:12000?Action=GetResources&SystemName=Answerbank&Type=question_
equivalence_class
```

The following example deletes a question equivalence class.

```
Action=ManageResource&SystemName=AnswerBank
data={
   "operation":"delete",
   "type":"question_equivalence_class",
   "ids": [
      "2012912839742797651"
   ]
}
```

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

Undelete a Question or Question Equivalence Class

After you delete a question or question equivalence class, there is a short period in which you can undelete it before it is permanently deleted from the system. For more information about how long an item is available after you delete it, see Modify the Expiration Time, below.

To undelete a question or question equivalence class, you use a `ManageResources` action in a POST request method, with the information about the item that you want to undelete provided in the `Data` parameter as a JSON object.

You must include the ID of the item that you want to undelete, and set the type of the operation to `question` or `question_equivalence_class`, as appropriate.

You can retrieve the ID of the question or question equivalence class by sending a `GetResources` action. For example:

```
http://localhost:12000?Action=GetResources&SystemName=Answerbank&Type=question_
equivalence_class
```

The following example restores a question equivalence class:

```
Action=ManageResource&SystemName=AnswerBank
data={
   "operation":"undelete",
   "type":"question_equivalence_class",
   "ids": [
      "2012912839742797651"
   ]
}
```

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

Modify the Expiration Time

When you delete a question or question equivalence class, Answer Server uses `ExpireDateType` fields in the Answer Bank Agentstore component to manage the expiration of deleted items.

All items in the Answer Bank Agentstore have an `ExpireDateType` field. Normally, these fields are set such that the questions and question equivalence classes never expire. When you delete an item, Answer Server sets this expiration time to a short time after you delete it.

By default, the expiration time is ten minutes. The default Answer Bank Agentstore configuration runs an expiration schedule every hour. You can therefore expect your question and question equivalences classes to be available to undelete for between ten minutes and an hour and ten minutes.

- To change the expiration time that Answer Server assigns to deleted items, modify the `UndeleteLifetime` configuration parameter in the section of the Answer Server configuration file where you configure the Answer Bank system. For more information, refer to the *Answer Server Reference*.
- To change the expiration schedule, modify the `ExpireTime` configuration parameter in the `[Schedule]` section of the Answer Bank Agentstore configuration file. For more information refer to the *IDOL Content Component Reference*.

Check the Status of an Update

The `GetJobStatus` action allows you to check the status of jobs in your `ManageResources` actions.

For example:

```
action=GetJobStatus&SystemName=MyAnswerBank&JobTokens=41,42,43
```

This action returns the status of the jobs with job IDs 41, 42, and 43.

For full details about the parameters available in the `GetJobStatus` action, refer to the *Answer Server Reference*.

Store Statistics for Your Answer Bank

You can configure Answer Server to store some information about the usage of your Answer Bank question equivalence classes in an ODBC-compatible database.

When you configure statistics storage, Answer Server stores popularity information for your question equivalence classes. Each time the text of an Ask action matches a question equivalence class, Answer Server updates the database.

Periodically, Answer Server also queries the database for the popularity information, and uses it to update a count field in the question equivalence class. You can use this field to sort question equivalence classes by popularity in the `GetResources` action.

To configure Answer Bank to store statistics

1. Open your configuration file in a text editor.
2. Find the configuration section for the Answer Bank system for which you want to store statistics.
3. In this configuration section, set `StatsStorage` to the name of a configuration section for the statistics database details. For example:

   ```
   [MyAnswerBank]
   StatsStorage=MyStatsDB
   ```

4. (Optional) Set `UpdatePopularityInterval` to the number of seconds between updates of the question popularity field in the Answer Bank Agentstore.

   By default, Answer Server updates the field every 600 seconds (10 minutes).

You can update the field more frequently if you need up-to-date information to sort by popularity. However, for performance reasons Micro Focus recommends that you do not update the popularity count field too frequently, because it might result in a large number of indexing operations in the Answer Bank Agentstore component.

5. (Optional) Set `PopularityWindowDays` to the number of days for which you want to consider popularity information for your question equivalence classes in the Answer Bank Agentstore.

   By default, Answer Server stores the count for the last seven days. That is, each time Answer Server updates the popularity field, it stores the number of times the question equivalence class was matched in the last week.

6. Create a configuration section for the statistics database. The name of this section must match the section you defined in the `StatsStorage` parameter in step 3. For example:

   ```
   [MyStatsDB]
   ```

7. In the statistics database configuration section, set `ConnectionString` to the connection string to use to connect to the statistics database. Answer Server passes this string on to the database, so you can use a connection string that works in any SQL client application. For example:

   ```
   [MyStatsDB]
   ConnectionString=Driver=PostgreSQL ANSI(x64); Server=sql-host.mycompany.com;
   Port=5432; Database=statsdb; Uid=postgres;password=password;
   ```

   The database that you configure must exist, but Answer Server creates all the tables that it needs to store the statistics information. The database must also be able and configured to accept dates in ISO-8601 formats (that is, `YYYY-MM-DD hh:mm:ss`).

   > **NOTE:**
   > On Linux operating systems, remove the spaces in the connection string.

8. (Optional) Set `Enabled` to **`True`**. The default value for this parameter is **`True`**, but you might want to set it explicitly for clarity. You can set `Enabled` to **`False`** if you want to stop storing the statistics.

The following example shows the complete configuration for this statistics database.

```
[MyAnswerBank]
StatsStorage=MyStatsDB
UpdatePopularityInterval=600
PopularityWindowDays=7

[MyStatsDB]
Enabled=True
ConnectionString=Driver=PostgreSQL ANSI(x64); Server=sql-host.mycompany.com;
Port=5432; Database=statsdb; Uid=postgres;password=password;
```

For more information about these configuration parameters, refer to the *Answer Server Reference*.

Retrieve the Information Stored in an Answer Bank

The `GetResources` and `GetStats` actions allow you to retrieve the information in your Answer Bank systems.

GetResources

The `GetResources` action allows you to retrieve the questions, answers, question equivalence classes, schemas, and XSDs for your answer systems. You can retrieve information for all systems by type, or you can restrict to a subset of systems or resource IDs, or exclude a particular set of IDs. You can also apply a filter to restrict the results to those that match some text, or a particular question state.

For full details about the parameters available in the `GetResources` action, refer to the *Answer Server Reference*.

Examples

The following action retrieves the schemas to use in a `ManageResources` action:

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=schema
```

The following action retrieves the first ten question equivalence classes stored in the specified Answer Bank system.

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=question_
equivalence_class
```

The following action retrieves the questions in the specified Answer Bank system that contain the keyword *President* in the question text or rule.

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=question&Fi
lter=%7B%20%22text%22%3A%20%22president%22%20%7D
```

The following action retrieves the questions in the specified Answer Bank system that have not been answered (`incoming`, `answerable`, and `needs_answer` states).

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=question&Fi
lter=%7B%20%22state%22%3A%20%5B%22incoming%22%2C%20%22answerable%22%2C%20%22needs_
answer%22%5D%20%7D
```

The following action returns the question equivalence class with ID `4371920660452849522`.

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=question_
equivalence_class&IDs=4371920660452849522
```

The following action returns the response XML Schema Definitions (XSDs) for the Answer Server actions:

```
http://localhost:12000?Action=GetResources&SystemName=MyAnswerbank&Type=XSD
```

For more examples of how to use `GetResources` to find particular information, see Manage an Answer Bank, on page 40.

GetStats

The `GetStats` action returns information about the number of questions and question equivalence classes with each question state (incoming, answered, and so on). For example, you can use this action to find out if you have any unanswered questions in the system.

For full details about the parameters available in the `GetStats` action, refer to the *Answer Server Reference*.

Example

```
action=GetStats&SystemName=MyAnswerBank
```

Chapter 6: Set Up a Fact Bank System

A Fact Bank system requires a Fact Store subcomponent, which is usually a SQL database. You can also configure Fact Bank to use a Lua script to retrieve facts from an external fact store.

Fact Bank also requires question parser Eduction grammar files, and a set of coding files. You can optionally configure additional Eduction grammars for advanced time normalization. Advanced time normalization extracts dates and times in various formats from questions and normalize them to a consistent format, to improve fact retrieval.

You must download and install the Fact Store component separately, and update the configuration with any Answer Server-specific configurations.

The following sections describe how to configure and set up your Fact Store and Eduction grammars, and how to configure the coding files.

- Configure the Fact Bank System 57
- Configure the Fact Store 60
- Create Coding Files 67
- The Question Parser Eduction Grammar 69
- Configure Security in Fact Bank 73

Configure the Fact Bank System

The Answer Server configuration file contains information about the subcomponents in your Fact Bank systems.

For any Fact Bank system, you must configure the name of your Question Parser and time normalization Eduction grammars and Lua scripts, and the locations of the Fact Bank coding files. You must also configure the Fact Store.

You can optionally also configure a second Eduction grammar for advanced time normalization. Advanced time normalization extracts dates and times in various formats from questions and normalize them to a consistent format, to improve fact retrieval.

The Fact Store component stores the facts that you want to be able to retrieve. The Fact Store content is structured data, containing the entities and properties for your facts. As such, Micro Focus recommends that you use a SQL database as the backend component for your fact store, because it is optimized for querying structured content.

Alternatively, you can use a Lua script to retrieve facts from an external fact store.

You specify the type of Fact Store that you want to use by configuring the `BackendType` configuration parameter in the system configuration section in the Answer Server configuration file. Depending on which back end type you choose, you must also set additional configuration parameters.

The following sections describe how to configure the Fact Bank system in Answer Server, depending on the type of Fact Store you choose.

DEPRECATED:
In earlier versions of Answer Server you could use an IDOL Content component fact store. This option is deprecated in Answer Server version 11.5 and later. Micro Focus recommends that you use a SQL database as your fact store.

The IDOL Content component fact store option is still available for existing implementations, but it might be incompatible with new functionality. For information about how to set up the IDOL Content component fact store, refer to the documentation for Answer Server version 11.4.

Configure a Fact Bank with a SQL Database Fact Store

The following procedure describes the Answer Server configuration you need to set up a Fact Bank system with a SQL database back end. For information about how to set up the SQL database, see Set Up a SQL Backend as Fact Store, on page 60.

To configure the Fact Bank System for a SQL database backend

1. Open the Answer Server configuration file in a text editor.
2. Find the `[Systems]` section, or create one if it does not exist. This section contains a list of systems, which refer to the associated configuration sections for each system.
3. After any existing systems, add an entry for your new Fact Bank system. For example:

   ```
   [Systems]
   0=MyAnswerBank
   1=MyFactBank
   ```

4. Create a configuration section for your Fact Bank system, with the name that you specified. For example, `[MyFactBank]`.
5. Set `Type` to **`FactBank`**.
6. Set `BackendType` to **`sqldb`**.
7. Set `ConnectionString` to the connection string to use to connect to the RDBMS that contains the fact store content.
8. Set `EductionQuestionGrammars` to the name of your question parser grammar. You can also optionally set `EductionEntities` to a list of entities to use from the specified grammars.
9. Set `EductionTimeGrammars` to the name of the Eduction grammar to use for advanced time normalization. You can also optionally set `TimeEntities` to a list of entities to use from the specified grammars.
10. Set `EductionLuaScript` and `TimeLuaScript` to the file name and path to the Lua scripts to use for question parsing and time normalization.
11. Set `CodingsPath` and `CodingsDatPath` to the locations of the coding files.
12. Save and close the configuration file.
13. Restart Answer Server for your changes to take effect.

For example:

```
[MyFactBank]
Type=factbank
// Question Parser
```

```
EductionQuestionGrammars=FactBankEductionGrammar.ecr
EductionTimeGrammars=datetime_processing.ecr
EductionLuaScript=lua/FactBankEduction.lua
TimeLuaScript=lua/DateTime.lua
// Fact Store
BackendType=sqldb
ConnectionString=Driver={PostgreSQL};Server=sql-
host.mycompany.com;Port=5432;Database=factstoredb;Uid=postgres;password=password;
// Coding Files
CodingsPath=./codings
CodingsDatPath=./codings
```

Configure a Fact Bank to Call a Lua Script

The following procedure describes the Answer Server configuration you need to set up a Fact Bank system to call a Lua script to retrieve facts. For information about how to write the script, see Use a Lua Scripts to Retrieve Facts, on page 65 and the *Answer Server Reference*.

To configure the Fact Bank system for a Lua Script backend

1. Open the Answer Server configuration file in a text editor.
2. Find the `[Systems]` section, or create one if it does not exist. This section contains a list of systems, which refer to the associated configuration sections for each system.
3. After any existing systems, add an entry for your new Fact Bank system. For example:

   ```
   [Systems]
   0=MyAnswerBank
   1=MyFactBank
   ```

4. Create a configuration section for your Fact Bank system, with the name that you specified. For example, `[MyFactBank]`.
5. Set `Type` to **`FactBank`**.
6. Set `BackendType` to **`Lua`**.
7. Set `ScriptPath` to the path to the Lua script that you want to run as your Fact Bank. Answer Server calls out to this script whenever it processes an `Ask` action that includes the Lua Fact Bank system, and returns the response as an answer.
8. (Optional) Set `ScriptFunction` to the entry function in your Lua script that Answer Server must call. The default entry function is called `fetch`.
9. Set `EductionQuestionGrammars` to the name of your question parser grammar. You can also optionally set `EductionEntities` to a list of entities to use from the specified grammars.
10. Set `EductionTimeGrammars` to the name of the Eduction grammar to use for advanced time normalization. You can also optionally set `TimeEntities` to a list of entities to use from the specified grammars.
11. Set `EductionLuaScript` and `TimeLuaScript` to the file name and path to the Lua scripts to use for question parsing and time normalization.
12. Set `CodingsPath` and `CodingsDatPath` to the locations of the coding files.

13. Save and close the configuration file.
14. Restart Answer Server for your changes to take effect.

Configure the Fact Store

The Fact Store component stores the facts that you want to be able to retrieve. The Fact Store content is structured data, containing the entities and properties for your facts. As such, Micro Focus recommends that you use a SQL database as the backend component for your fact store, because it is optimized for querying structured content.

Alternatively, you can use a Lua script to retrieve facts from an external fact store.

The following sections describe how to set up the Fact Store backend.

> **DEPRECATED:**
> In earlier versions of Answer Server you could use an IDOL Content component fact store. This option is deprecated in Answer Server version 11.5 and later. Micro Focus recommends that you use a SQL database as your fact store.
>
> The IDOL Content component fact store option is still available for existing implementations, but it might be incompatible with new functionality. For information about how to set up the IDOL Content component fact store, refer to the documentation for Answer Server version 11.4.

Set Up a SQL Backend as Fact Store

You can use a SQL database to store facts and qualifiers for your Fact Store.

Answer Server can connect to any RDBMS that supports SQL. You specify how to connect to the database by setting the appropriate connection string in the `ConnectionString` configuration parameter in the FactBank configuration. The most fully tested options are:

- SQLite
- PostgreSQL

The SQL Fact Store has two required tables, one for facts, and one for qualifiers. The facts table stores the values of entity properties. The qualifiers table stores the names and values of particular sets of qualifiers associated with the properties. Each table includes qualifier combination values, which link the properties to the associated qualifiers.

In addition, there are two optional tables, one for sources and one for security types. The sources table stores information about the sources of your facts, including optional security information to allow you to restrict the facts by user permissions. The security types table stores the security configuration information for your security types.

Your Answer Server installation includes PostgreSQL schema files for these tables, and a utility script that you can use to apply these schemas to a database by using the psql utility. These files are available in the `/factbank/schemas/postgresql` directory in your installation.

The following sections describe the tables in more detail, and provide some best practices for how to organize your fact stores.

Manage Your Tables

Micro Focus recommends that you organize your tables by creating a separate database for each set of facts and qualifiers. In this case, each database is the backend for its own Fact Bank system, which optimizes the database queries required for an `Ask` action.

For example, if you have a collection of facts about company sales histories, and a collection of facts about the products that a company offers, you might create a `sales` database and a `products` database. Each database is a separate Fact Bank instance in your Answer Server setup, and you can easily query one or both, as required.

Facts Table

The facts table stores the values of entity properties.

This table must have the name `facts`. The facts table has four required columns, and one optional column, described in the following table.

Column	Type	Description
`entity_code`	text	The code for the entity that this row is about, from your `entity_to_code.txt` coding file.
`property_code`	text	The code for the property that this row is about, from your `property_to_code.txt` coding file.
`property_value`	text	The value of this property for the specified entity, in the associated qualifier combination.
`qualifier_combination`	integer	The reference value for the rows in the `qualifiers` table that contain qualifiers that apply to a particular property value. This value corresponds to the values in the `qualifier_combination` column in the `qualifiers` table. Answer Server uses this column to find the appropriate property, entity, or property value when a question contains a particular qualifier. If this value does not correspond to a value in the qualifiers table, Answer Server treats it as having no qualifiers. Micro Focus recommends that you reserve a value to use for properties that do not have qualifiers (usually **0**).
`source_id`	integer	(Optional) The reference value for the row in the `sources` table that contains the source for this fact. This value corresponds to the values in the `id` column in the `sources` table. Answer Server uses this column to find and return the source for a particular fact. If this value is missing, or does not correspond to a value in the sources table, Answer Server returns the source as **SQLDB**. Micro Focus recommends that you reserve a value to use for properties that do not have source information (usually **0**).

You can optimize the performance of the facts table by creating indexes on each column. For example, in a PostgreSQL instance, Micro Focus recommends that you create a btree index on each column.

Qualifiers Table

The qualifiers table stores the codes and values for qualifiers, and the qualifier combination reference that links a qualifier to a row in the `facts` table.

The table must have the name `qualifiers`. The qualifiers table has three columns, described in the following table.

Column	Type	Description
`qualifier_combination`	integer	The qualifier combination reference that identifies qualifiers that are associated with a particular property value for an entity.
`qualifier_code`	text	The code for the property that this qualifier is about, from your `property_to_code.txt` file
`qualifier_value`	text	The value of the qualifier in this qualifier combination

Most data sets will have multiple rows with the same qualifier combination. For example, if your facts table contains the ares of different types of land in a country over time, you might have something like the following table for qualifiers.

qualifier_combination	qualifier_code	qualifier_value
1	LANDTYPE	Farmland
1	YEAR	2015
2	LANDTYPE	Forest
2	YEAR	2015
3	LANDTYPE	Farmland
3	YEAR	2016
4	LANDTYPE	Forest
4	YEAR	2016

In this case, the qualifier combination 1 relates to farmland in 2015, qualifier combination 2 relates to forest in 2015, and so on.

You can optimize the performance of the qualifiers table by creating indexes on each column. For example, in a PostgreSQL instance, Micro Focus recommends that you create a btree index on each column.

Sources Table

The sources table is an optional table to allow you to store the sources for your facts. You can use this option with the `source_id` column in the facts table to store the details of the fact sources. Answer Server returns the source information with the fact when it returns in an `Ask` action.

The table must have the name `sources`. The sources table has two required columns and two optional columns, described in the following table.

Column	Type	Description
`id`	integer	A primary key integer ID value for the source.
`source`	text	The name of the source. Answer Server returns this value in the `Ask` action when it returns a fact that has the associated source ID.
`acl`	text	(Optional) The Access Control List (ACL) for this source. You can use this option if you want to use user security for your facts.
`security_type_id`	integer	(Optional) The reference value for the row in the `security_types` table that contains the security type for this source. This value corresponds to the values in the `id` column in the `security_types` table. You can use this option if you want to use user security for your facts. Answer Server uses this column to find and return the security type for a particular source. If this value does not correspond to a value in the `security_types` table, Answer Server treats it as having no security. You can set `security_type_id` to `NULL`, or you might want to reserve a value to use for properties that do not have security (usually **0**).

Security_Types Table

The security types table is an optional table to allow you to store the security type information for your sources. You can use this option with the `security_type_id` column in the sources table to store the details of the security types.

The table must have the name `security_types`. The security types table has two columns, described in the following table.

Column	Type	Description
`id`	integer	A primary key integer ID value for the security type. This value corresponds to the IDs that you use in the `sources` table.
`friendly_name`	text	The name of the security type. This value must correspond to the security type configuration section in your Answer Server configuration file.

SQL Fact Store Example

The following section describes an example of setting up a SQL backend Fact Bank.

A company wants to make its exports data available for question answering. It exports two products, Psi and Omega, to two countries, Germany and France.

In 2015, Psi cost €5 and Omega cost €20. The company exported:

- 50 units of Psi and 20 units of Omega to Germany.
- 10 units of Psi and 30 units of Omega to France.

In 2016, Psi cost €7 and Omega cost €18. Sales increased, and the company exported:

- 80 units of Psi and 100 units of Omega to Germany.
- 40 units of Psi and 50 units of Omega to France.

The following tables show the Fact Bank **Facts** and **Qualifiers** table for this information.

Facts Table

entity_code	property_code	property_value	qualifier_combination
QPSI	REXPORTS	50	0
QPSI	REXPORTS	10	1
QPSI	REXPORTS	80	2
QPSI	REXPORTS	40	3
QOMEGA	REXPORTS	20	0
QOMEGA	REXPORTS	10	1
QOMEGA	REXPORTS	80	2
QOMEGA	REXPORTS	40	3
QPSI	RPRICE	5	4
QOMEGA	RPRICE	20	4
QPSI	RPRICE	7	5
QOMEGA	RPRICE	18	5

Qualifiers Table

qualifier_combination	qualifier_code	qualifier_value
0	S.COUNTRY	Germany
0	S.YEAR	2015
1	S.COUNTRY	France
1	S.YEAR	2015
2	S.COUNTRY	Germany
2	S.YEAR	2016
3	S.COUNTRY	France
3	S.YEAR	2016
4	S.YEAR	2015
5	S.YEAR	2016

The following example shows the entity code values in the `code_to_entity.txt` file:

```
QPSI=Psi
QOMEGA=Omega
```

The following example shows the property code values in the `code_to_property.txt` file:

```
S.YEAR=Year,string
S.COUNTRY=Country,string
RPRICE=Price,string
REXPORTS=Exports,string
```

You can use this Fact Bank setup to answer questions such as:

- *What were the 2015 exports of Psi?*
- *How many Omega were exported to Germany?*
- *What is the average price of Omega?*

Your Answer Server installation includes example files that allow you to test this setup. The example files are included in the `sql_example` directory.

To use this example, you must have a PostgreSQL instance (running on *`PostgresHost`* and *`PostgresPort`*, with a user who has access to a `factbank_example` database.

To set up the example

1. Restore the database from the factback_example.pgdump file into your PostgreSQL instance, by using a command of the following form:

   ```
   psql -h PostgresHost -p PostgresPort -U UserName factbank_example < factbank_
   example.pgdump
   ```

2. In your Answer Server configuration file, configure a Fact Bank system to use the plain text coding files provided in the example, by setting the `CodingsPath` parameter to point to the example coding files. For example:

   ```
   [MyExampleFactBank]
   Type=factbank
   CodingsPath=C:\AnswerServer\sql_example
   ...
   ```

3. In the Fact Bank system configuration, set the `ConnectionString` to the appropriate connection string for your example database.
4. In the Fact Bank system configuration, set the `EductionQuestionGrammars` parameters to the location of your `FactBankEductionGrammar.ecr`, which is distributed with Answer Server.
5. Save and close the configuration file.
6. Start all the Answer Server components.

You can now use the `Ask` action to ask the questions listed above and to check the responses.

Use a Lua Scripts to Retrieve Facts

You can configure a Fact Bank system in Answer Server that calls a Lua script to get facts. Answer Server calls out to this script whenever it processes an `Ask` action that includes the Lua Fact Bank

system, and returns the response as an answer.

For example, you might have an external data source or API that contains the factual information. Rather than convert the information into a Fact Store database format, you can use a Lua script to retrieve the information directly.

> **NOTE:**
> When Answer Server processes a question in an `Ask` action, it might find multiple ways to parse the question. This might mean that Answer Server calls your script multiple times for a single `Ask` action, although some of the parsings might not match any available facts.

To use a Lua script, you must ensure that the Fact Bank configuration has the `BackendType` configuration parameter set to **`Lua`**. See Configure a Fact Bank to Call a Lua Script, on page 59.

Create a Fact Retrieval Script

The Lua script must implement a script function that Answer Server can call. By default, Answer Server calls the `fetch` function, but you can use a different name and set the `ScriptFunction` configuration parameter to the appropriate function name.

The script function must accept a single `LuaIncompleteFact` object, and return an array of `LuaCodifiedFact` objects.

For example, the following fetch function always returns an empty array:

```
function fetch(incomplete)
  return {}
end
```

The `LuaIncompleteFact` object represents the question that has been asked. It has at most one target element missing, which the Lua script must attempt to find the information for.

The `LuaIncompleteFact` object also includes:

- an entity, which is the object of the question (unless the question target is the entity itself, in which case the entity value is empty).
- zero or more `LuaIncompleteFactProperty` objects, which define a property of the question. Properties might be negated (that is, you want to find facts that do not match the property).

  Each `LuaIncompleteFactProperty` object can also contain zero or more `LuaIncompleteFactQualifier` objects, which define a qualifier for the property. Qualifiers might be negated (that is, you want to find facts where the property does not match the qualifier).

You initialize the `LuaCodifiedFact` object from a `LuaIncompleteFact` object by using the `initToCodifiedFact` method. The `LuaCodifiedFact` object must not have any missing elements (for example, it must have an entity value).

The `LuaCodifiedFact` can also include zero or more `LuaCodifiedFactProperty` objects, which specify a property of the fact. Each `LuaCodifiedFactProperty` object can also have zero or more `LuaCodifiedFactQualifier` objects.

Entities, properties, and qualifiers are all coded on entry into the function, and decoded before Answer Server returns the information to the end user.

The Lua script has access to Lua functions and methods that are included in many IDOL products, as well as functions and methods that are specific to Answer Server. For full details and examples of the available functions, refer to the *Answer Server Reference*.

Create Coding Files

The coding files are simple files that describe the entity, property, and qualifier codes in your Fact Bank system. It also defines any aliases for any of the entities, properties, and qualifiers, and maps all aliases to the same code.

A Fact Bank system requires four coding files:

- `code_to_property.txt`. Assigns a unique code to each property and qualifier in your data, as well as a canonical human-readable name and the data type.
- `property_to_code.txt`. An inverse mapping of the property and qualifier codes, including any aliases.
- `code_to_entity.txt`. Assigns a unique code to each entity in your data (that is, things for which you might want to know the values of a property).
- `entity_to_code.txt`. An inverse mapping of the entity codes, including any aliases.

The following sections use a simple example to show how to create the coding files from your data.

Example Data

The example data starts with facts, organized in a table. This version uses CSV format:

```
product_name, color, buy_price, sell_price, sold_last_year
alpha, red, 10, 12, 3500000
beta, blue, 11, 13, 2000000
gamma, green, 9, 10, 1000000
```

For this example, you might want to be able to answer questions such as:

- What is the purchase price for alpha?
- What was the selling price of beta?
- What color is gamma?

Generate the Property Code Files

The properties in your data are the values that you want to find in the Fact Bank. For a table like the one in this example, the properties are the columns in the table.

The `code_to_property.txt` coding file assigns a unique code for each property and qualifier. This coding file also defines the canonical human-readable name for the property or qualifier, and sets its type. You can use the following types:

- `string`. The property or qualifier values are strings.
- `time`. The property or qualifier values are times in the ISO format `YYYY-MM-DDTHH:NN:SS`.

- `entity`. The property or qualifier values are entity codes. In this case, you must list the entity code in the `code_to_entity.txt` file, and you must list the possible values for this entity in the `entity_to_code.txt` file. This option allows you to map multiple values to the same qualifier code.

> **NOTE:**
> If your data values contain punctuation characters, such as commas (`,`) and equals signs (`=`), you must percent-encode the value in the coding files. For example, use `%3D` for an equals sign.

For example, the following sample is the `code_to_property.txt` file for the example data in the previous section.

```
PRODUCT_NAME=product,string
COLOR=color,string
BUY_PRICE=buying price,string
SELL_PRICE=selling price,string
SOLD_LAST_YEAR=sold last year,string
```

The `property_to_code.txt` coding file contains the inverse mapping of the `code_to_property.txt` file, without the type information. You can also include aliases for a value, on a separate line.

For example, the following sample is the `property_to_code.txt` file for the example data. It includes the alias *sale price* for the `SELL_PRICE` code.

```
product=PRODUCT_NAME
color=COLOR
buying price=BUY_PRICE
purchase price=BUY_PRICE
selling price=SELL_PRICE
sale price=SELL_PRICE
sold last year=SOLD_LAST_YEAR
```

Generate the Entity Code Files

The entities are the things that you want to find the property values for. For the example table, the obvious choice is the `product_name`.

The `code_to_entity.txt` coding file assigns a unique code to each entity.

> **NOTE:**
> If your data values contain punctuation characters, such as commas (`,`) and equals signs (`=`), you must percent-encode the value in the coding files. For example, use `%3D` for an equals sign.

For example, the following sample is the `code_to_entity.txt` for the example data.

```
ALPHA=alpha
BETA=beta
GAMMA=gamma
```

The `entity_to_code.txt` coding file contains the inverse mapping of the `code_to_entity.txt`. You can also include aliases for the entity names, on a separate line.

For example, the following sample is the `entity_to_code.txt` for the example data. It includes the alias *alpha one* for the `ALPHA` code.

```
alpha=ALPHA
beta=BETA
gamma=GAMMA
alpha one=ALPHA
```

Generate the Fact Store Data

This section describes how to convert a table of information into Fact Store content, for a SQL database backend type. It uses the example table specified in the Create Coding Files, on page 67, and also assumes you have set the entity and property codes as described in that section.

```
product_name, color, buy_price, sell_price, sold_last_year
alpha, red, 10, 12, 3500000
beta, blue, 11, 13, 2000000
gamma, green, 9, 10, 1000000
```

Create a Fact Store Table for a SQL Database

In the SQL database backend, each property and property value for a particular entity is a row in the `Facts` table. For more information about the format of this table, see Set Up a SQL Backend as Fact Store, on page 60.

The following table shows the table rows for the alpha product, assuming that all these properties refer to the same qualifier combination.

entity_code	property_code	property_value	qualifier_combination
ALPHA	COLOR	red	0
ALPHA	BUY_PRICE	10	0
ALPHA	SELL_PRICE	12	0
ALPHA	SOLD_LAST_YEAR	35000000	0

For this data, the qualifiers you use in the `qualifiers` table might include a year, or a selling region. For example:

qualifier_combination	qualifier_code	qualifier_value
0	YEAR	2015
0	REGION	Americas

The Question Parser Eduction Grammar

The Fact Bank Question Parser is an IDOL Eduction grammar designed for question answering, which the Answer Server internal Eduction module uses to parse questions. This special grammar file defines many different forms that questions can take, and extracts the *entities*, *properties*, and *qualifiers* in the questions.

In general, an entity is the topic of the question, or a topic in your Fact Store data. Entities have properties, which define pieces of information that you might want to find, or which you might want to use to find a particular entity. A qualifier is a piece of information that modifies the property.

For example, in the question *What is the population of the USA in 1850*, you might define:

- USA to be the entity.
- population to be the property.
- 1850 to be a qualifier.

Many questions can be interpreted in more than one way, depending on how you set up your data.

The Fact Bank grammar can find the entities, properties, and qualifiers in a variety of different question formations.

When the Question Parser processes a question, the grammar might find several valid interpretations. In this case, the Question Parser returns all the options to Answer Server, which attempts to find the associated entities, properties, and qualifiers in your coding files, and then in your data.

The values that match in your Fact Store at this stage depend on how you have set up and stored your data.

> **NOTE:**
> For time and entity type answers, Answer Server merges duplicate answers when multiple interpretations return the same value.

The examples in the sections below demonstrate many of the different forms of questions in English that the Question Parser can detect. The questions are examples, and the list is not exhaustive. In addition, some of the forms are not mutually exclusive (that is, Question Parser might detect both forms). In these cases, the correct form depends on your data.

Processors

The Q&A grammar also detects a number of processors, which affect the question in a similar way to qualifiers, but which might require further calculation on the data. The following processors are supported (processors listed on the same line are equivalent):

- min, minimum
- max, maximum
- mean, average
- total, sum
- first, oldest
- last, latest

Example Questions

The following section lists several example questions in forms that the Question Parser can parse, and which Answer Server can use to retrieve the relevant facts from the Fact Store.

In the questions, entities are specified in *italics*, properties are specified in **bold**, and qualifiers are underlined.

Questions to Find a Property Value

The following questions show forms where the Question Parser detects an entity and property name, and any associated qualifiers. For these questions, Answer Server attempts to find the property value in the Fact Store.

- What is the **population** of *France*?
- What is *France*'s **population**?
- Can you tell me the *French* **population**?
- The **population** of *France* is what?
- How many **people** live in *France*?

  This question also matches the property **number of people** (the grammar automatically constructs this property because of the *how many* in the question).
- What was the **population** of *France* in 2010?
- What is the female **population** of *France*?
- What is the non-female **population** of *France*?

  In this example, the qualifier female is negated, so Answer Server finds facts that do not have this qualifier.
- What is the **population** of *France* that is female?
- What is the **population** of *France* that isn't female?

  In this example, the qualifier female is negated, so Answer Server finds facts that do not have this qualifier.
- What did *Dickens* **write** in 1837?
- Who **discovered** *America* in 1492?

  This question also matches the property **discovered by** (the grammar automatically constructs this property because of the *who* in the question).

Questions to Find Processed Property Value

The following questions show forms where the Question Parser detects an entity and property name, and a processor that qualifies the value that the question requests. For these questions, Answer Server attempts to find the property value in the Fact Store.

Processors are marked in `monospace` font.

- What is the `latest` **population** of *Russia*?
- What is the `maximum` male **population** of *Russia*?
- When was *George Washington* `first` **elected**?
- What are the `total` **electoral votes** of *George Washington*?

Questions to Find the Name of an Entity

The following questions show forms where the Question Parser detects a property and one or more property values. For these questions, Answer Server attempts to find the name of an entity in the Fact Store.

Property values are marked in ***bold italic***.

- Where is ***Paris*** the **capital** of?
- Whose **capital** is ***Paris***?
- Which country has a **capital** that is ***Paris***?
- Who was **president** in ***1810***?
- Who was **president** of ***America*** in ***1810***?
- Who are the **presidents** of ***America*** who are fictional?

  In this example, the fictional qualifier modifies the property **presidents**, which has the value ***America***.
- Who are the **presidents** of ***America*** who are ***male***?

  In this example, ***male*** is treated as the value of a separate unnamed property of the entity that the question requests.
- What country **borders** ***France*** and ***Spain***?
- What ***country*** **borders** ***Portugal***?

Questions to Find the Value of a Qualifier

The following questions show forms where the Question Parser detects an entity, a property, and a property value. For these questions, Answer Server attempts to find the value of a qualifier that matches these values.

- When did the *USA* have ***George Washington*** as **president**?

  This question matches a special point in time qualifier.
- When did *George Washington* become ***president***?

  This question matches the special **position held** property, and a special start time qualifier.

Modify the Question Parser Eduction Grammar

The default question parser grammar provides a large number of possible questions in English (see Example Questions, on page 70). Versions of this grammar file are also available in French, German, Italian, and Spanish.

However, if you want to use a Fact Bank in a different language, or if you find that the results for a particular question that you want to use are not ideal, you can add your own entities to the Fact Bank User Grammar.

The `FactBankUserGrammar.xml` file is included in your Answer Server installation, with the other Fact Bank resources. To modify the Question Parser grammar, you must modify this XML file and compile it by using the edktool command-line tool, which is included in the Eduction SDK.

The `FactBankUserGrammar.xml` has comments that explain in more detail how to modify the grammar. You can add new entities to the file, for example to provide question formats in different languages. It includes the content in the existing `FactBankEductionGrammar.ecr` file, so your modifications extend the existing grammar.

When you have modified the grammar, you compile it and deploy it in Answer Server, replacing the standard `FactBankEductionGrammar.ecr` file.

For more information about Eduction grammars and edktool, refer to the *Eduction SDK Programming Guide*.

Configure Security in Fact Bank

You can use IDOL mapped security with Fact Bank, by including the ACL information in your SQL backend fact store.

To use IDOL mapped security with Fact Bank you must:

- configure the appropriate security types in your Answer Server configuration file.
- uses a `sources` table in your SQL database fact store, with the ACL values.
- add a `security_types` table in your SQL database fact store, with the details of the security types.

Configure the Security Types in Answer Server

Security configuration in Answer Server is similar to the configuration in the IDOL Content component. Depending on your data sources, it is likely that your Answer Server security configuration contains the same security types as your IDOL Content component data store.

You do not need any additional security libraries to run security in Answer Server fact bank.

To configure security in Answer Server

1. Open your configuration file in a text editor.
2. Find the `[Security]` configuration section, or add one if it does not exist.
3. In the `[Security]` section, set the `SecurityInfoKeys` parameter to specify the security encryption keys to use to encrypt and decrypt the security information. You can set the `SecurityInfoKeys` parameter either to the name of an AES key file (recommended) or to a comma-separated list of four signed 32-bit integers. For information about how to generate a key file with the `autpassword` command-line tool, refer to the *IDOL Server Administration Guide*.

   For example:

   ```
   [Security]
   SecurityInfoKeys=MyAESKeyFile.ky
   ```

4. In the `[Security]` section, list the security types that you want to use.

   ```
   0=NT
   1=Netware
   2=Notes
   3=Exchange
   ```

5. Create a section for each of the security types you defined (the section must have the same name as the security type). For each section, provide settings that determine how Answer Server handles that security type. For example:

   ```
   [NT]
   Type=AUTONOMY_SECURITY_V4_NT_MAPPED

   [Netware]
   Type=AUTONOMY_SECURITY_NETWARE_MAPPED
   ```

```
[Notes]
Type=AUTONOMY_SECURITY_V4_NOTES_MAPPED

[Exchange]
Type=AUTONOMY_SECURITY_EXCHANGE_MAPPED
```

6. Save and close the configuration file.

Set Up Fact Store Tables for Security

To use mapped security with your facts, you must use the `sources` table in your SQL fact store.

The sources table contains details of the source for your facts. Each source has an ID, which you use in the facts table to identify the source for an individual fact. For example:

Facts Table

entity_code	property_code	property_value	qualifier_combination	source_id
COMPANY	CEO	Jane Smith	0	1

Sources Table

id	source
1	http//www.example.com/about

When you use mapped security, the sources table also includes an access control list (ACL) for each source, and a security type ID. The security type ID corresponds to a row in the `security_types` table, which contains details of your configured security types.

For example:

Sources Table

id	source	acl	security_type_id
1	http//www.example.com/about	1:U:G:NU:9sjbyPA,9tnU38jBwfA:NG:9sjbxMHOwt/d8A	3

Security_Types Table

id	friendly_name
1	Exchange
2	Netware
3	NT

For more information about the tables in your SQL fact store, see Set Up a SQL Backend as Fact Store, on page 60.

Chapter 7: Set Up a Passage Extractor System

This section describes how to set up a Passage Extractor system, and configure the subcomponents.

- Configure the Passage Extractor System 75
- Train Passage Extractor Classifiers 76
- Entity Extraction in Passage Extractor 81
- Troubleshoot Passage Extractor 86

Configure the Passage Extractor System

The Answer Server configuration file contains information about the subcomponents in your Passage Extractor systems.

For any Passage Extractor system, you must configure the host and port details of your data store, which is an IDOL Content component that contains the documents that Answer Server uses to find answers. For entity extraction, you must also configure the details for your Eduction grammars, and the Passage Extractor Agentstore component.

Passage extractor also uses question classifiers, to determine the type of a question, and therefore what entities to extract from candidate answers. The classifier is required. To configure one, you must train a classifier, which you can optionally save to use in future sessions.

The following procedure describes how to configure the Passage Extractor system in Answer Server.

For more details about the configuration parameters for the Passage Extractor system, refer to the *Answer Server Reference*.

To configure the Passage Extractor System

1. Open the Answer Server configuration file in a text editor.
2. Find the `[Systems]` section, or create one if it does not exist. This section contains a list of systems, which refer to the associated configuration sections for each system.
3. After any existing systems, add an entry for your new Passage Extractor system. For example:

   ```
   [Systems]
   0=MyAnswerBank
   1=MyFactBank
   2=MyPassageExtractor
   ```

4. Create a configuration section for your Passage Extractor system, with the name that you specified. For example, `[MyPassageExtractor]`.
5. Set `Type` to **`PassageExtractor`**.
6. Set `IDOLHost` and `IDOLACIPort` to the host name and ACI Port of the IDOL Content component that contains the documents that you want to use to find answers.
7. Set `AgentstoreHost` and `AgentstoreACIPort` to the host name and ACI Port of the IDOL Content component that contains entity agents.

8. Set `EductionGrammars` to a comma-separated list of the Eduction grammars to use for entity extraction.
9. If you want to save the question classifier files, set `ClassifierFile` to the location and name of the file to use to save the classifier file, and set `LabelFile` to the location and name of the file to use to save the label file.

   > **TIP:**
   > The Answer Server installation ZIP package includes a default set of classifier training files. To use these files, set `ClassifierFile` to the location of the `en_svm.dat` file, and set `LabelFile` to the location of the `en_labels.dat` file.

10. Save and close the configuration file.
11. Restart Answer Server for your changes to take effect.

For example:

```
[MyPassageExtractor]
Type=PassageExtractor
// Data store IDOL
IdolHost=localhost
IdolAciport=6002
// Entity Agentstore
AgentStoreHost=localhost
AgentStoreAciport=5002
// Eduction
EductionGrammars=pe_grammar.ecr,NUM.ecr,HUM.ecr,date_eng.ecr,money_eng.ecr
// Classifier Files
ClassifierFile=en_svm.dat
LabelFile=en_labels.dat
```

Train Passage Extractor Classifiers

The Answer Server Passage Extractor uses a question classifier to determine what type a question is, and therefore what entities (if any) to extract from candidate answers. The type refers to the type of information that the question is requesting. For example, the question *How many points make up a perfect fivepin bowling score?* is looking for a number, while the question *What is an annotated bibliography?* is looking for a description.

The question classifier is always required. The Passage Extractor system does not return any answers without it.

To configure the question classifier, you can train one, or use a saved one from a previous training run.

> **NOTE:**
> You can save classifiers only if you set the `ClassifierFile` and `LabelFile` configuration parameters in your Passage Extractor system configuration. See Configure the Passage Extractor System, on the previous page.

The Answer Server installation ZIP package includes a default set of classifier files passage extractor installation.

To use the pre-trained classifier, copy the DAT files from the installation ZIP package to a suitable location. Then update the `ClassifierFile` and `LabelFile` configuration parameters to point to the `en_svm.dat` and `en_labels.dat` files respectively. See Configure the Passage Extractor System, on page 75.

The following sections provide more information about how to create and train your own classifiers.

Create a Training File

For the English language, the Answer Server installation includes a suitable training file. For other languages, you might need to create your own training file to describe the kind of question classifications that you expect to send to your Passage Extractor.

Each line of the training file defines a label and an example question, in the following format:

```
Label;Example Question
```

The example questions are the training. The label specifies the kind of information that the question is requesting. For example, the first few lines of the training file might be:

```
DESC:desc;What did the only repealed amendment to the U.S. Constitution deal with?
NUM:count;How many points make up a perfect fivepin bowling score?
DESC:def;What is an annotated bibliography?
NUM:date;What is the date of Boxing Day?
```

The default training file uses a Text Retrieval Conference (TREC) classification system to specify question classifiers. Micro Focus recommends that you use this classification system, which is based on a commonly used set. For more information, see Training File Labels, on the next page. However, you can use your own classification system if required.

Train a Classifier

To train the question classifier, you use the `ManageResources` action, which accepts a JSON object with the details of the training file. For example:

```
action=ManageResources&SystemName=passageextractor&Data=JSON
```

Where the JSON object takes the following form:

```
{
   "operation": "train",
   "type": "classifier",
   "trainingfile": "classifier_training.txt",
   "savemodel": "true"
}
```

If you do not want to save the training model (for example, during testing), set `savemodel` to **`false`**. You cannot set `savemodel` to **`true`** unless you have configured the `ClassifierFile` and `LabelFile` parameters for your Passage Extractor system.

The `trainingfile` parameter sets the location and name of a suitable training file. The training file contains a set of training questions, and a label that specifies the sort of answer that the question is looking for (for example, a person, place, or description).

You can use the `GetResources` action to retrieve the whole JSON schema for the operation, in the same way as for Answer Bank systems. See Find the JSON Schema for Your Update, on page 41.

Classifier Behavior File

In addition to the main classifier and label files, there is a classifier behavior file, which is available in the Answer Server installation.

The classifier behavior file contains details of question classifications that it must treat differently. In particular, it includes information about whether to always or never consider other question classifications when a particular classification is identified as the primary classification.

For example, you generally want to consider other location classifications when a question matches the `LOC:other` classification. Similarly, for classifications that match descriptive questions you can explicitly never include other classifications, because classifications that match entities are less relevant, but might score higher in the results.

The primary classification is determined by a probability threshold, which is 0.85 by default.

If you move or rename the classification behavior JSON file, modify the `ClassifierBehaviorFile` configuration parameter to specify the new name and location.

Training File Labels

The label has two parts, separated by a colon. The first part is the class of the question, the second part is a subdivision. There are six classes:

- Abbreviation (`ABBR`). Questions concerning abbreviations.
- Entity (`ENTY`). Questions about entities (things).
- Description (`DESC`). Questions that ask for a definition or a description.
- Human (`HUM`). Questions about people or organizations.
- Location (`LOC`). Questions about places.
- Numeric (`NUM`). Questions about numbers.

The following table describes the available subdivisions for each class.

Label	Description	Example question
Abbreviation		
`ABBR:abb`	Abbreviation questions.	What is the abbreviation for micro?
`ABBR:exp`	Expression questions.	What does HPE stand for?
Entity		
`ENTY:animal`	Animal questions.	Which type of bird migrates between the North and South pole?

Label	Description	Example question
ENTY:body	Organs of the body.	Which artery takes blood to the head?
ENTY:color	Colors.	What was FDR's favorite color?
ENTY:cremat	Inventions, books, other creative pieces.	What film starred Al Pacino and Robert Deniro?
ENTY:currency	Currency names.	What currency do they use in Laos?
ENTY:dismed	Diseases and medicine.	What diseases can be transmitted by mosquitoes?
ENTY:event	Events.	What major sporting event was held in Australia in 2000?
ENTY:food	Food.	What Italian dish is made of soft dumplings?
ENTY:instrument	Musical instrument.	What is Jimi Hendrix famous for playing?
ENTY:lang	Languages.	What language is spoken in Cambodia?
ENTY:letter	Letters like a-z.	Which letter is the most common in Finnish?
ENTY:other	Other entities.	Which shape has 14 sides?
ENTY:plant	Plants.	What is the most poisonous plant?
ENTY:product	Products.	What shampoo is best for dandruff?
ENTY:religion	Religions.	What is the religion that worships Prince Phillip?
ENTY:sport	Sports.	Which sport involves people dressed in white standing around doing absolutely nothing for several days straight?
ENTY:substance	Elements and substance.	What chemicals make up mica?
ENTY:symbol	Symbols and signs.	What is the chemical formula for diamond?
ENTY:techmeth	Techniques and methods.	What methods are used to measure atmospheric pressure?
ENTY:termeq	Equivalent terms.	What is the name of the Thai alphabet?
ENTY:vehicle	Vehicles.	What is the largest plane ever made?
ENTY:word	Words with a special property.	What English words have Japanese origin?
Description		
DESC:def	A definition of something.	What is dyslexia?

Label	Description	Example question
DESC:desc	A description of something.	What is the difference between a centipede and a millipede?
DESC:manner	A manner of action.	How do I apply for a driving license?
DESC:reason	Reasons.	What caused the American civil war?
Human		
HUM:gr	A group or organization of persons.	Which body elects the president?
HUM:ind	An individual.	Who wrote 'The unbearable lightness of being?'
HUM:title	The title of a person.	What is his position in the company?
HUM:desc	A description of a person.	Who is Serena Williams?
Location		
LOC:city	Cities.	What is the capital of France?
LOC:country	Countries.	Which country is the best governed?
LOC:mount	Mountains.	What is the highest mountain in Vietnam?
LOC:other	Other locations.	What is the biggest lake in the world?
LOC:state	States.	Which state has the highest lowest point?
Numeric		
NUM:code	Postal codes or other codes.	What is the White House's post code?
NUM:count	The number of something.	How many pillars of faith are there in Islam?
NUM:date	Dates.	When was the battle of Waterloo?
NUM:dist	Linear measures.	How long is the River Nile?
NUM:money	Prices.	How much is that doggie in the window?
NUM:ord	Ranks.	Which episode of the Star Wars saga has the cutest aliens?
NUM:other	Other numbers.	What is Stephen Hawking's IQ?
NUM:period	The duration of something.	How long since the last ice age?

Label	Description	Example question
NUM:perc	Fractions and percentages.	What proportion of the Earth is covered in water?
NUM:speed	Speeds.	How fast is the speed of light?
NUM:temp	Temperature.	What is the boiling point of nitrogen?
NUM:volsize	Size, area, and volume.	How big is the sea?
NUM:weight	Weight.	How much does the average human weigh?

Entity Extraction in Passage Extractor

Passage Extractor uses entity extraction to provide more concise, specific answers. It attempts to find the shortest possible section of a document that answers the original question. Depending on the question, the answer might be a single word or name, or a few sentences of description.

There are two types of entity extraction that you can use:

- **Eduction**. IDOL Eduction provides a set of grammars, which define the entities that you want to find. This method is very powerful for pattern matching, and finding entities that match a particular structure, such as phone numbers. Answer Server includes an embedded IDOL Eduction module for entity extraction.
- **Agent matching**. The Passage Extractor Agentstore component stores agents that define entities in the Agentstore component. Passage Extractor sends any candidate answers in an agent query, which returns the matching entity agents. This method is very powerful for entities that have a clearly defined value, such as names.

To get the most out of Passage Extractor, you must configure at least one of an Eduction grammar or Agentstore component for entity extraction. You can use both to make the most out of the different methods, and to get the best answers.

Configure the Passage Extractor Agentstore

The IDOL Agentstore component is a specially configured IDOL Content component.

In agent search, you send plain text or a document to the Agentstore, which returns any agents that match the document. In Passage Extractor, you store entity agents in the Agentstore . For example, each entity agent might define a single name (perhaps with one or more alias names).

When you ask a question, Passage Extractor finds candidate answers in your data store, and uses an agent search to find the entities that these candidate answers contain.

Configure the Agentstore Component

The Answer Server package includes a predefined Passage Extractor Agentstore configuration file, as well as several predefined IDX documents that you can optionally use to populate your Agentstore with entities.

> **NOTE:**
> You must use a separate Agentstore component for Passage Extractor and Answer Bank systems.

To configure the Agentstore component for your Passage Extractor

1. In your Answer Server installation directory, copy the Agentstore `agentstore.cfg` configuration file, and the IDX files.
2. Open your Agentstore installation directory.
3. Paste the Answer Server Agentstore configuration file and IDX files. Overwrite the installed configuration file (you might want to make a copy of it first).
4. Open the configuration file in a text editor.
5. Update the `[License]` section with the host and port information for your License Server. For more information, see Configure the License Server Host and Port, on page 21.
6. In the `[Server]` section, find the `Port` parameter. Check that the specified port is available on the host machine, or change it to an available port.

   > **NOTE:**
   > If you modify the port, make sure to update the system configuration in your Answer Server configuration file. See Configure the Passage Extractor System, on page 75.
7. In the `[Service]` section, find the `ServicePort` parameter. Check that the specified port is available on the host machine, or change it to an available port.
8. Save and close the configuration file.

Index Entity Agents

After you have configured the Agentstore, you must index the entity agents that Passage Extractor uses.

To do this, you can use a `DREADD` index action to add the IDX files that you copied into the Agentstore installation directory. For example:

```
http://localhost:5001/DREADD?C:\AnswerServer\passageextractor\agents-HUM_ind.idx.gz
```

Customize Entity Extraction

The Passage Extractor entity extraction file provides Answer Server with a map to specify what components to use to extract entities, depending on the question classification.

When you ask a question, Passage Extractor classifies it by using the question classifier, and then finds matching documents and document sections in the data store. It uses IDOL Content highlighting to find the most relevant passages, which it uses as candidate answers. Passage Extractor then uses Eduction and an Agentstore component to find entities in the candidate answers that match the question classification.

For example, if you have an Agent entity database with the names of plants, and you send a question that Passage Extractor classifies as plants, Passage Extractor uses the Agentstore component to find the relevant plant entities in the candidate answer text.

By default, if you configure an Agentstore component, Passage Extractor uses the Agentstore for the classifications `HUM:gr`, all `LOC` classifications, `ENTY:plant`, `ENTY:animal`, and `ENTY:lang`. It uses Eduction and Agentstore for the `HUM:ind` question classification, and Eduction only for all other question classifications.

You can use the Entity Extraction file to modify these classifications, for example if you create additional Agent entity files for your data.

> **NOTE:**
> You do not need to specify an entity type to extract for every question classification. If a question classification does not appear in the entity extraction file, Passage Extractor does not attempt to extract entities. This might be appropriate for many question classifications (for example, if the appropriate answer is a long description, there might not be a corresponding entity).

Passage Extractor also attempts to corroborate the candidate answers, by comparing how often particular entities occur. In most cases, this improves the quality of the result answers.

In some cases, corroboration might not be appropriate. For example, if valid answers include very common words (such as *one* and *two*), the words might occur in multiple places, and be falsely corroborated as a likely answer. For this reason, corroboration is turned off for the `NUM:count` entity type in the default entity extraction JSON file.

You might also want to turn corroboration off if likely answers occur only once in your data set. In these cases you can modify the entity extraction JSON file to turn corroboration off for particular entities.

The Entity Extraction File Format

The entity extraction file contains the question classifications, which match the values that you use in the classifier training file. For each question classification, it also contains at least one of:

- a list of Eduction entities that Passage Extractor must use to find entities for the question classification.
- a list of Agentstore databases that Passage Extractor must query to find entities for the question classification.

When there is an Agentstore database, you can also specify Agent `FieldText` to use in a query to the Agentstore entity database for the question classification.

The entity extraction file is a JSON file, with the following structure:

```
{
   "entity_map": [
      {
         "entity_type": "QuestionClass1",
         "agentstore": {
            "databases": [ListOfAgentstoreDatabases],
            "fieldtext": "FieldTextRestriction"
         },
         "eduction": {"entities": [ListOfEductionEntities]},
         "corroborate": Boolean
      },
      {
```

```
         "entity_type": "QuestionClass2",
         "agentstore": {
            "databases": [ListOfAgentstoreDatabases],
            "fieldtext": "FieldTextRestriction"
         },
         "eduction": {"entities": [ListOfEductionEntities]},
         "corroborate": Boolean
      }
      ...
   ]
}
```

where,

QuestionClassN	is the name of the question classification (for example, `HUM:ind`).
ListofEductionEntities	is an array of relevant Eduction entities.
ListOfAgentstoreDatabases	is an array of databases in the Agentstore component that contain relevant entities.
FieldTextRestriction	is an IDOL `FieldText` expression to use to restrict the Agent query in the specified database.

You must specify at least one of the `eduction` or `agentstore` properties for each question classification. If you specify the `agentstore` property, the `database` property is required, but `fieldtext` is not.

If you do not want to use entity extraction for a particular question classification, do not include it in the entity extraction file.

The `corroborate` property is optional. The default value is `true`.

The following example gives some of the question classifications in the default entity extraction file:

```
{
   "entity_map": [
      {
         "entity_type": "HUM:ind",
         "agentstore": {"databases": ["people"]},
         "eduction": {"entities": ["hum/ind"]}
      },
      {
         "entity_type": "NUM:date",
         "eduction": {"entities": ["num/date", "date/*"]}
      },
      {
         "entity_type": "ENTY:plant",
         "agentstore": {
            "databases": ["organisms"],
            "fieldtext": "MATCH{PLANTAE,VIRIDIPLANTAE}:ORGANISMS_KINGDOM"
         }
      },
```

```
    {
       "entity_type": "NUM:count",
       "eduction": {"entities": ["num/count"]},
       "corroborate": false
    },

...
```

Modify the Entity Extraction File

The default entity extraction file, included in your Answer Server installation , is appropriate for most installation. However, you might need to modify the file if:

- you do not want to configure an Agentstore component for entity extraction, and want to use Eduction entity extraction for those question classifications instead.
- you create additional Agent entity files and index them into your Passage Extractor Agentstore.
- you create custom Eduction entities that you want to use for entity extraction.
- you want to define entity extraction for additional question classifications
- you want to turn off corroboration for some question classifications.

To update the entity extraction file

1. Open the entity extraction JSON file in a text editor.
2. Make the necessary modifications. You can add, delete, or update, any of details for the question classifications.

   To turn off corroboration, add the `corroborate` property in a particular group and set it to `false`. For example:

   ```
   {
      "entity_type": "NUM:count",
      "eduction": {"entities": ["num/count"]},
      "corroborate": false
   }
   ```
3. Save and close the entity extraction file.
4. Restart Answer Server for your changes to take effect.

> **NOTE:**
> If you add new question classifications that do not exist in the classifier training file, you must also update the classifier training file and retrain the classifier. See Train Passage Extractor Classifiers, on page 76.

Use a Different Entity Extraction File

You can use the `EntityExtractionFile` configuration parameter to configure the location of the entity extraction file. If you want to move or rename the entity extraction file, or use a different file for any reason, you must modify the value of this parameter to specify the name and location of the new file.

Troubleshoot Passage Extractor

This section contains some information about how to check that Passage Extractor is working correctly.

Passage Extractor Does Not Return Any Answers

- Check that you have a question classifier trained and configured. Answer Server writes an error to the application log if there is no available classifier. See Train Passage Extractor Classifiers, on page 76.
- Check that the host and port details for the subcomponents are correct. In particular, check that the IDOL Content component that you use for the data store is configured correctly. See Configure the Passage Extractor System, on page 75.

Passage Extractor Returns an Answer that is Incorrect or Unusual

- Increase the log level for your Answer Server to `FULL`, by modifying the log stream configuration. See Customize Logging, on page 31.

  When the log level is set to `FULL`, Answer Server generates a lot of extra information about how Passage Extractor found its answers, including the candidate passages, extracted entities, and scoring information.

  To improve answers in these cases, you might need to add additional entities, or rephrase the question. You might also want to check the data in your IDOL Content component data store, to see whether your data set contains appropriate content for the answer.

Chapter 8: Ask Questions in Answer Server

This section describes how to ask questions and get answers from Answer Server.

- Ask a Question 87

Ask a Question

To get answers from Answer Server, use the `Ask` action. Specify the question text in the `Text` parameter. For example:

```
http://localhost:12000?Action=Ask&Text=Who won the award for Best Picture in 2012?
```

The `Ask` action also has a number of optional parameters to allow you to tune your results.

- `SystemNames` restricts the `Ask` action to particular configured systems. For example:

  ```
  http://localhost:12000?Action=Ask&Text=Who won the award for Best Picture in
  2012?&SystemNames=MovieFactBank,Answerbank
  ```

  By default, Answer Server sends the question to all configured systems, in the order in which they are configured in the `[Systems]` section. If you set `SystemNames`, Answer Server sends the question only to the specified systems, in the order in which you specify them.
- `MaxResults` specifies the maximum number of results to retrieve.

  ```
  http://localhost:12000?Action=Ask&Text=Who won the award for Best Picture in
  2012?&SystemNames=MovieFactBank,Answerbank&MaxResults=2
  ```

  In this case (for default sorting), if Answer Server finds the required number of results from the first system, it does not request answers from further systems.
- `Sort` specifies how to sort the results. By default, Answer Server sorts answers in the order of the system that the answer comes from (either the configured order, or the order in `SystemNames`). You can set `Sort` to **`Confidence`** instead, to sort the results by score.

  ```
  http://localhost:12000?Action=Ask&Text=Who won the award for Best Picture in
  2012?&MaxResults=2&Sort=Confidence&SystemNames=MovieFactBank
  ```

  > **NOTE:**
  > Each answer system in Answer Server scores its answers independently, so scores from one system might not be comparable to scores from another.

- `MinScore` specifies the minimum score that an answer must have for it to return as a result.
- `CustomizationData` specifies additional information to include in the request. For a fact bank or passage extractor system, you can use this option to include a security info string. In this case, fact bank returns only facts that the user has access to, and passage extractor returns answers only if they appear in documents that the user has access to.

  ```
  Action=Ask&Text=What is the gift and entertainment policy?&CustomizationData=
  [{"system_name":"MyPassageExtractor", "security_info":
  "MTQ0lGDBkNrJvBv0pOi+QDBK1z6y/1/09BqL4Vu/18W7JGCy8Pvm4/wixO/pI99/A=="}]
  ```

For more information about the `Ask` action, refer to the *Answer Server Reference*.

Chapter 9: Set Up a Conversation System

This section describes how to set up a Conversation system, and configure the subcomponents.

- Configure the Conversation System 89
- Create a Task Configuration File 90
- Configure the Conversation Agentstore 122

Configure the Conversation System

The Answer Server configuration file contains information about the subcomponents and settings for your Conversation systems.

For any Conversation system, you must configure the name of a JSON task configuration file, which defines the conversation task in detail. For details about how to create this file, see Create a Task Configuration File, on the next page.

The following procedure describes how to configure the Conversation system in Answer Server.

In addition, you can optionally configure session expiration for your conversation sessions, and settings for an IDOL Agentstore component that contains triggers for your conversation sessions.

For more details about the configuration parameters for the Conversation system, refer to the *Answer Server Reference*.

To configure the Conversation System

1. Open the Answer Server configuration file in a text editor.
2. Find the `[Systems]` section, or create one if it does not exist. This section contains a list of systems, which refer to the associated configuration sections for each system.
3. After any existing systems, add an entry for your new Conversation system. For example:

   ```
   [Systems]
   0=MyAnswerBank
   1=MyFactBank
   2=MyPassageExtractor
   3=MyConversation
   ```

4. Create a configuration section for your Conversation system, with the name that you specified. For example, `[MyConversation]`.
5. Set `Type` to **`Conversation`**.
6. Set `TaskConfigurationFile` to the path and file name of the JSON file that contains your task configuration file. You can also specify a comma-separated list of path and file names if you have split your task configuration across multiple files.
7. Optionally set any additional settings for your conversation system. For more information, refer to the *Answer Server Reference*.

8. Save and close the configuration file.
9. Restart Answer Server for your changes to take effect.

For example:

```
[MyConversation]
Type=Conversation
TaskConfigurationFile=C:\AnswerServer\Conversation\tasks.json
// Trigger Agentstore
AgentStoreHost=localhost
AgentStoreAciport=5002
// Session Expiration
SessionExpirationIdleTime=600
SessionExpirationInterval=60
```

Create a Task Configuration File

The Conversation task configuration file describes all the settings for a particular conversation task, including prompts, triggers, response validation, and the details of Lua scripts to run.

You can retrieve the full schema for the task configuration JSON file by sending the `GetResources` action with the `Type` parameter set to **`Schema`**.

> **TIP:**
> For ease of maintenance, you can split your task configuration across multiple JSON files. Answer Server merges the configurations together when it runs.

The task configuration file contains one or more conversation tasks. Each task must contain a unique ID (`id`). You can also set a `display_id`, which is a user friendly display name that Answer Server uses in a disambiguation message when user input matches the triggers for more than one task (see Conversation Triggers, on page 92).

The other options you use in a task depend on what you want to the task to do. The following sections describe the main options that you can use in task configuration:

- Pre-Task Actions, on the next page. Pre-task actions run at the start of the task. You can use pre-task actions to set an initial prompt for the task, by setting some text or by specifying a Lua function to run to generate a prompt.
- Conversation Triggers, on page 92. A conversation trigger defines when to run a particular conversation task. You can use several different types of trigger to activate a particular task by matching user text against a simple word list, a regular expression pattern, or an agent.
- Task Requirements, on page 98. Task requirements request pieces of information that the user must provide before Answer Server can complete the tasks. You can save the requirement result as a variable, either for the duration of the task or for the whole conversation session.
- Post-Task Actions, on page 108. Post-task actions run at the end of the task, after all the requirements have been satisfied. You can use post-task actions to send an acknowledgment to confirm that the requirements are met, or to call a Lua function to perform an external operation.
- Task Routing, on page 110. Routing allows you to specify the task to run after the current task. You can route directly to a particular task, or you can use a further prompt to allow you to route to another task depending on user input.

- Response Validation, on page 103. Response validation allows you to check that the user provides a response that you can use in the task, and to deal with the response if it is not valid. You can validate the response by using a simple word list, a regular expression, Eduction, or a Lua script.
- Lua Processing Scripts, on page 113. You can use Lua functions in your task configurations to run additional more complicated operations, and to send calls to external systems to complete tasks. You configure a single Lua script in your task configuration, and then call individual functions. You can use functions to generate a custom preamble for a task, to perform an operation at the end of the task, and you can use Lua functions for response validation and to process user input that is not valid.

In addition to the task configuration, and the associated validator configuration, the task configuration file also contains:

- Initial Task, on page 115. The ID of the first task to run when the conversation session starts, if you do not have any initial user text. You must include an initial task in your task configuration.
- Fallback Task, on page 115. The ID of a task to run when there are no other active tasks. You must include a fallback task in your task configuration.
- Default Messages, on page 116. Custom text to use for default responses, such as when the user does not provide any valid input, or to provide the user with disambiguation options when their text triggers more than one task. There are default values for these messages, which you can override with your own values (for example, if you want to create conversations in a different language).
- Task Cancellation, on page 120. Settings that determine how to allow users or the system to cancel a task, and what to do when a task is canceled.

Pre-Task Actions

Pre-task actions run before the main part of the task. When user input triggers a particular task, Answer Server runs the pre-task actions before checking for requirements. The main use of this option is to provide an initial prompt at the start of the task.

To configure pre-task actions, you set the `pre` object in the configuration object for an individual task.

The `pre` object is not required. If you do not add a `pre` object to the task, Answer Server runs the next step of the task, for example processing and gathering the requirements.

The following table describes the properties that you can set in the `pre` object.

Property	Type	Description
`response`	string	(Optional) A string response to return to the user. For example, this might be an introductory sentence to start the task.
`lua`	string	(Optional) A lua function to run before the rest of the task, for example to generate a dynamic task preamble response. The function that you specify must accept a `taskUtils` object. See Lua Processing Scripts, on page 113.

```
{
   "tasks" : [
      {
         "id" : "GREET",
         "pre" : {
```

```
            "response" : "Hello and Welcome to the Virtual Assistant. How can I
help you?"
         },
         "trigger" : {
            "simple" : {
               "phrases" : [
                  "hello",
                  "hi"
               ]
            }
         }
      }
   ]
}
```

Conversation Triggers

A conversation trigger defines when to run a particular conversation task. When you start a conversation session, or when a user provides text, Answer Server finds the conversation trigger that matches the input, and runs the appropriate task.

You can use several different types of triggers:

- **Agent**. Trigger a task based on a match against your Conversation Agentstore index. To use this option you must configure an IDOL Agentstore component in your Conversation system configuration. See Configure the Conversation Agentstore, on page 122.
- **Regular Expressions**. Trigger a task based on a text match against a regular expression pattern.
- **Simple**. Trigger a task based on a simple exact phrase match.

You can configure different tasks with different types of triggers.

> **TIP:**
> When a user sends some text that matches more than one trigger, Answer Server returns a disambiguation prompt. This prompt uses the `display_id` in your tasks to display the options to the user. See Task Disambiguation, on page 97.

Not all tasks require a conversation trigger. For example, the tasks that you configure as the initial task and fallback task might not need a trigger (see Default Tasks, on page 115). Similarly, you might have tasks that run only when called by explicit routing from another task.

The number of tasks that you add triggers to depends on your conversation structure and routing.

To configure a trigger, you set the `trigger` property in the configuration object for an individual task, which contains different properties depending on the type of trigger you use.

The following sections describe the types of conversation triggers in more detail, and provide examples of how to configure them.

Agent Triggers

> **NOTE:**
> To use agent triggers, you must configure an IDOL Agentstore component in your Conversation system configuration.

Agent triggers use IDOL agent search to find matching triggers. In agent search, you send plain text or a document to the Agentstore, which returns any agents that match the input text. In the case of agent triggers, Answer Server sends the user input text to the Agentstore, which returns any matching agent triggers.

You link the value of an Agentstore field in the trigger agent (usually the document reference) to a task in your task configuration JSON file. When the user text matches a particular agent trigger, Answer Server runs the corresponding task for that agent.

> **TIP:**
> By default, Answer Server uses the document reference field in your agents as the task ID. You can set the `AgentstoreReferenceField` configuration parameter in your Conversation system configuration to change the field that Answer Server uses as the task ID.

Agent triggers use IDOL text search to match user text. For example, it includes IDOL stemming to allow more flexible term matching, and you can use term weighting to influence how triggers match. For more information, refer to the *IDOL Server Administration Guide*.

To configure an agent trigger, you add a `trigger` object with the `agent` property, set to the document reference for the agent that triggers the task.

For example:

The following IDX document represents a simple agent trigger, which you index into the Conversation Agentstore:

```
#DREREFERENCE LunchConversationTrigger
#DREDATE 2017/07/01
#DRETITLE
Lunch
#DRECONTENT
food lunch sandwich
#DREDBNAME MyAgentDB
#DREENDDOC
```

The following JSON shows a task that uses this agent as a trigger.

```
{
   "tasks" : [
      {

         "id" : "LUNCH",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
```

```
            "agent" : "LunchConversationTrigger"
         }
      }
   ]
}
```

This task runs when a user sends any text that matches the terms *food*, *lunch*, or *sandwich* in IDOL text matching.

Regular Expressions Triggers

Regular expressions triggers match user text against a regular expression pattern that you provide in the task configuration. Answer Server runs the task when the user input text matches your regex.

To configure a regular expressions trigger, you add a `trigger` object with the `regex` property. The `regex` property contains a `pattern` subproperty, which you must set to a regular expression pattern in ECMAScript regular expression format.

For example:

```
{
   "tasks" : [
      {
         "id" : "LUNCH",
         "display_id" : "order lunch",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
            "regex" : {
               "pattern" : "(Book|order) .* lunch"
            }
         }
      }
   ]
}
```

You can also provide a weight for the trigger by setting the `weight` property in the trigger object. The weight value is a number between one and 100. The default value is 100. Answer Server uses this weight to determine the best triggers to return in a disambiguation response. See Trigger Options, on page 98.

The weight value is constant for a particular trigger. You can assign a higher weight to triggers that have more restrictive trigger conditions. For example, if a user must match a more complex regular expression to trigger a task, it is more likely to be the task that they want, and so you assign a higher weight.

For example:

```
 "trigger" : {
   "regex" : {
      "pattern" : "(Book|order) .* lunch"
   },
```

```
   "weight" : 80
}
```

Simple Triggers

Simple triggers match user text against a list of phrases that you provide in the task configuration. Answer Server runs the task when the user input text matches one of the phrases you provide.

In a simple trigger, you must provide a list of phrases. You can optionally also provide a required prefix and suffix. By default, matches are case sensitive, but you can also use a property in your configuration to specify that matches must be case insensitive.

To configure a simple trigger, you add a `trigger` object with the `simple` property. The following table describes the subproperties that you can set in the `simple` object.

Property	Type	Description
`phrases`	array, strings	(Required) A list of phrases that you want to match. The task runs if the user input text matches at least one of the specified strings. The array must have at least one item.
`prefix`	string	(Optional) A string that must occur at the start of the user input text for this trigger to match.
`suffix`	string	(Optional) A string that must occur at the end of the user input text for this trigger to match.
`case_insensitive`	Boolean	(Optional) Set to **true** to match user text case insensitively. The default value is **false** (case sensitive matching).

For example:

```
{
   "tasks" : [
      {
         "id" : "LUNCH",
         "display_id" : "order lunch",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
            "simple" : {
               "prefix" : "I want to",
               "phrases" : [
                  "order some lunch",
                  "buy a sandwich",
                  "book a table for lunch",
                  "order food"
               ]
            }
         }
      },
```

```
      {
         "id" : "BREAKFAST",
         "display_id" : "order breakfast",
         "pre" : {
            "response" : "I can help you order some breakfast."
         },
         "trigger" : {
            "simple" : {
               "phrases" : [
                  "get breakfast",
                  "order breakfast",
                  "eat breakfast",
                  "buy breakfast"
               ],
               "case_insensitive" : true
            }
         }
      }
   ]
}
```

The first of these example tasks matches user text that starts with *I want to* and continues with one of the listed phrases, such as *I want to order some lunch*, or *I want to order a sandwich*. The matching for this task is case sensitive.

The second task matches user text that contains any of the listed phrases, and matches case insensitively.

You can also provide a weight for the trigger by setting the `weight` property in the trigger object. The weight value is a number between one and 100. The default value is 100. Answer Server uses this weight to determine the best triggers to return in a disambiguation response. See Trigger Options, on page 98.

The weight value is constant for a particular trigger. You can assign a higher weight to triggers that have more restrictive trigger conditions. For example, if a user must supply more matching keywords to trigger a task, it is more likely to be the task that they want, and so you assign a higher weight.

For example:

```
"trigger" : {
   "simple" : {
      "phrases" : [
         "get breakfast",
         "order breakfast",
         "eat breakfast",
         "buy breakfast"
         ],
      "case_insensitive" : true
   },
   "weight": 90
}
```

Task Disambiguation

In some cases, the user might provide text that matches the trigger for more than one task. In this case, Answer Server returns a disambiguation prompt, which allows the user to select the appropriate option from a list of tasks that match their input.

Answer Server uses the task `display_id` property in the prompt that it displays to the user. If you do not specify a `display_id`, it uses the `id` property.

For example:

```
{
   "tasks" : [
      {
         "id" : "LUNCH",
         "display_id" : "order lunch",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
            "simple" : {
               "prefix" : "I want to",
               "phrases" : [
                  "order food"
               ]
            }
         }
      },
      {
         "id" : "BREAKFAST",
         "display_id" : "order breakfast",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
            "simple" : {
               "phrases" : [
                  "get breakfast",
                  "order breakfast",
                  "eat breakfast",
                  "buy breakfast",
                  "order food"
               ]
            }
         }
      }
   ]
}
```

If a user sends the text *I want to order food*, both of these task triggers match. Answer Server then returns the text *Which of the following did you mean: order lunch, or order breakfast?* (you can modify this message, if required. See Default Messages, on page 116).

After the user selects the appropriate task, Answer Server processes the selected task. It sends the original text that activated the disambiguation prompt to the new task, so that the new task can use it for automatic requirement gathering, where appropriate (see Automatic Requirement Gathering, on page 101).

You can define the maximum number of triggered tasks that return in a disambiguation message by using the `trigger_options` configuration. See Trigger Options, below.

Trigger Options

The `trigger_options` object allows you to set additional properties that define how your task triggers work. You can optionally add this object at the top level of your task configuration file. The settings apply to all your tasks.

The following table describes the properties that you can set in the `trigger_options` object.

Property	Type	Description
`max_triggers`	integer	(Optional) The maximum number of triggers that the conversation task can return in a disambiguation response. Answer Server sorts the triggers by weight and returns up to `max_triggers` options with the highest weight. The default value is **6**. For non-agent triggers, you define the `weight` property in the `trigger` configuration object. For agent triggers, Answer Server uses the returned `autn:weight` value when it queries for matching agent triggers. Answer Server considers triggers with zero weight only if there are no matches with non-zero weights.
`weight_range`	number (0-100)	(Optional) The weight range for task triggers to allow in task disambiguation. When Answer Server returns a disambiguation response, it does not return any triggers that have a weight that is more than `weight_range` below the highest weight. By default, there is no limit.

For example:

```
{
   "tasks" : ...
   "trigger_options": {
      "max_triggers": 5,
      "weight_range": 50
   }
}
```

Task Requirements

Task requirements specify a piece of information that the user must provide before Answer Server can complete the task. Tasks can have multiple requirements.

When a user provides information to satisfy a requirement, Answer Server updates a variable. It stores the variable to use later in the task (for example to send information to an external system). By default, Answer Server stores the variable only for the duration of the current task. You can also choose to store a variable for the whole conversation session.

To configure requirements, you set the `requirements` property in the configuration object for an individual task. The requirements property takes an array of requirement objects. The following table describes the properties that you use to define a requirement.

Property	Type	Description
`id`	string	(Required) A unique ID for the requirement. Answer Server uses this value as the name of the task or session variable that it stores for this requirement.
`prompt`	string	(Required) The prompt to send to the user to request the information. You can include session and task variables, by inserting the variable ID in double curly brackets, for example `{{MYVARIABLE}}`. The variable must already be set in an earlier part of the task (for a task variable) or conversation session (for a session variable). You can optionally use the `task:` or `session:` prefix to specify the type of variable, for example `{{task:MYVARIABLE}}`. If you do not use a prefix, Answer Server searches for the variable in the task variables first, and then the session variables.
`scope`	enum, string	(Optional) The scope of the variable: `local` or `session`. The default value is `local`, which means that the variable is available only in the current instance of the current task. Set scope to `session` to store the variable for the whole conversation session.
`validation`	array, strings	(Optional) A list of validators to use to validate the user input. Specify the ID of each validator that you want to use to validate the current requirement. You define the validators separately in the `validators` section of the task configuration JSON file. See Response Validation, on page 103.
`prompt_required`	Boolean	(Optional) Set to **true** to require that the prompt for this requirement must be sent to the user before this requirement can be set. By default, Answer Server attempts to answer requirements from the trigger or subsequent user text, unless the requirement does not have any validators (in which case, Answer Server always presents the prompt, and the prompt for the next requirement). For more details, see Automatic Requirement Gathering, on page 101.
`ask_options`	object	(Optional) Options to use for the `Ask` action when the user text does not pass the requirement validators. By default, if the user text is not valid, the conversation system returns a default message. You can set `ask_options` if you want to treat user text that is not a valid answer for the requirement as a question to ask in your other answer systems. The `ask_options` JSON object can contain the following properties: • `max_results` (number) Required. The maximum number of results to

		retrieve from the other answer systems. • `systems` (string) Optional. The systems that you want to send the `Ask` action to.
`suggestions`	array, strings	(Optional) A list of suggested answers to return to the user with the requirement prompt. When you set this option, Answer Server returns the suggestions with the prompt in the `Converse` action response. You can use these values to present suggestions to your users. Answer Server does not validate the configured suggestions. However, it does validate the user responses that include them, so Micro Focus strongly recommends that you use valid values as suggestions. You can include session and task variables, by inserting the variable ID in double curly brackets, for example `{{MYVARIABLE}}`. The variable must already be set in an earlier part of the task (for a task variable) or conversation session (for a session variable). You can optionally use the `task:` or `session:` prefix to specify the type of variable, for example `{{task:MYVARIABLE}}`. If you do not use a prefix, Answer Server searches for the variable in the task variables first, and then the session variables.
`user_cancel`	object	(Optional) An object that defines keywords that a user can use to cancel a task, and the action to perform if they do. For details of the configuration properties, see User Cancellation, on page 120. The `user_cancel` options in the requirements section override any values that you set in the main task configuration or for the individual task.
`system_cancel`	object	(Optional) An object that defines what actions to take when the requirement receives multiple non-valid responses. For details of the configuration properties, see System Cancellation, on page 121. The `system_cancel` options in the requirements section override any values that you set in the main task configuration or for the individual task.

When Answer Server runs a task that has requirements, it returns the `pre` response for the task (if present), and then the prompt for the first requirement. Each prompt returns in a separate `<prompt>` tag in the same `Converse` action response, and you can choose how to display this information in your user interface.

For example:

```
{
   "initial_task" : "GREET",
   "tasks" : [
      {
         "id" : "GREET",
         "pre" : {
            "response" : "Hello and welcome to the Virtual Assistant. Before we get
started, I'd like to ask you a couple of questions."
         },
         "requirements": [
```

```
            {
               "id": "USER_NAME",
               "prompt": "What is your name?",
               "prompt_required": true,
               "scope": "session"
            },
            {
               "id": "USER_COUNTRY",
               "prompt": "What country do you live in?",
               "prompt_required": true,
               "validation": [ "VALIDATE_COUNTRIES" ]
            }
         ]
      }
   ],
   "validators": [
      {
         "id": "VALIDATE_COUNTRIES",
         "eduction": {
            "grammars": "place_countries.ecr"
         }
      }
   ]
}
```

When the task runs at the start of a conversation session, the user receives the following initial message:

Hello and welcome to the Virtual Assistant. Before we get started, I'd like to ask you a couple of questions.
What is your name?

When the user answers, the task stores the answer in the `USER_NAME` session variable and then returns the following message:

What country do you live in?

Answer Server validates the user input by using the `VALIDATE_COUNTRIES` validator. For more information about validator configuration, see Response Validation, on page 103. If the response is valid, Answer Server stores the value in the `USER_COUNTRY` task variable.

Automatic Requirement Gathering

The example in the previous section uses the `prompt_required` flag to ensure that Answer Server sends the prompt to the user, and the user must provide an answer to each requirement individually.

In many cases, you might want to automatically check the user response text for additional answers to your task requirements. For example, if you send a prompt to a user for their name, and they reply with their name and company, you might want to extract the company name from this text, rather than subsequently asking the user what company they work for.

Automatic requirement gathering applies only to requirements that have validators, and where the prompt is not required. When a user sends some text, Answer Server checks whether any portion of their input text matches the validator for the current requirement. If it does, it also checks whether the user input text matches the validators for any other requirements in the current tasks, and sets the appropriate task and session variables if it does.

> **NOTE:**
> For requirements that do not have validators (for example, those with free text input), Answer Server always presents the prompt. It also always presents the prompt for the following requirement, even if the user provided the information in an earlier response.

This process applies to the initial user text that triggers the task. If the trigger text contains the answer for the first requirement in the task, Answer Server also checks whether the text contains answers for other requirements.

For example:

```
{
   "tasks" : [
      {
         "id" : "LUNCH",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
            "regex" : {
               "pattern" : "(Book|order) .* lunch"
            }
         },
         "requirements": [
            {
               "id": "FOOD_TYPE",
               "prompt": "Do you feel like a sandwich or a panini?",
               "validation": [ "FOODTYPEVALIDATOR" ]
            },
            {
               "id": "FILLING",
               "prompt": "What filling would you like?",
               "validation": [ "FILLINGVALIDATOR" ],
               "suggestions": ["cheese", "ham", "turkey"]
            }
         ],
         "post" : {
            "response" : "We'll get you a {{FILLING}} {{FOOD_TYPE}} right away!"
         }
      }
   ],
   "validators" : [
      {
         "id" : "FOODTYPEVALIDATOR",
         "simple" : {
```

```
            "matches": [
               { "values": [ "sandwich", "panini" ] }
            ]
         }
      },
      {
         "id" : "FILLINGVALIDATOR",
         "simple" : {
            "matches": [
               { "values": [ "cheese", "ham", "turkey" ] }
            ]
         }
      }
   ]
}
```

If a user triggers this task with the text *I want to order some lunch*, Answer Server returns the prompt for the first requirement as usual. However, if the user triggers the task with the text *I want to order a cheese panini for lunch*, the trigger text provides the answer to both of the requirements. In this case, Answer Server responds with the initial task prompt (*I can help you order some lunch*), and the acknowledgment (*We'll get you a cheese panini right away*).

Related Topics

- Post-Task Actions, on page 108
- Response Validation, below
- Conversation Triggers, on page 92

Response Validation

Response validation allows you to check that the user response contains the information you need to complete a task, or route to an appropriate task. You use validators in the requirements for your tasks (see Task Requirements, on page 98).

You can use several different types of validation:

- **Simple**. Validate against a simple list of words and phrases, which the user input must match exactly.
- **Regular expression**. Validate against a regular expression pattern, which the user input must match.
- **Eduction**. Validate against a set of Eduction grammars and entities. This validator uses the embedded Answer Server Eduction module.
- **Lua**. Validate by using a Lua function. This validator calls a function in an external Lua script, which you must configure in the task configuration file. See Lua Processing Scripts, on page 113.

You configure validators in the `validators` section of the task configuration file. This property takes an array of validators. Each validator must contain a unique ID, and a configuration object (one of `simple`, `regex`, `eduction`, and `lua`, depending on the validation type). The following sections describe each type of validator configuration in more detail, and provide examples of each type of validator.

Each validator object can also optionally include the properties in the following table.

Property	Type	Description
`invalid_input_lua`	string	(Optional) A Lua function in the configured Lua script to call if the user input is not valid. For example, you might use this to submit the user response to your other answer systems. See Lua Processing Scripts, on page 113.
`inverted`	Boolean	(Optional) Set to **true** if you want to invert the match (that is, to consider user input text as valid if it does not match the validator). The default value is **false** (user input text must match the validator).

Simple Validation

In simple validation, you provide a simple list of values that the user input must match.

The following table describes the properties that you can set in the `simple` validation configuration object.

Property	Type	Description
`matches`	array, objects	One or more match JSON objects, which contain the following properties: • `values` (array, strings) Required. A list of values that you want to accept as user input for this validator. Answer Server accepts these values if they appear as part of a sentence in the input. • `return_value` (string) Optional. The value that the validator uses if the user input matches any of the values in the array. If you do not set this value, the validator returns the value that was matched in the user input.
`case_insensitive`	Boolean	(Optional) Set to **true** if you want to match values case insensitively. The default value is **false** (case sensitive matching).

For example:

```
{
   "tasks": [
      ...
   ]
   "validators": [
      {
         "id" : "UKcountryname",
         "simple" : {
            "matches": [
               {
                  "values": [ "United Kingdom", "UK", "Great Britain", "GB" ],
                  "return_value": "UK"
               },
               {
                  "values": [ "England", "Northern Ireland", "Scotland", "Wales" ]
               }
            ],
```

```
            "case_insensitive" : true
         }
      }
   ]
}
```

This example validator checks that the user input matches one of the listed names for the United Kingdom. If the user text includes one of *United Kingdom*, *UK*, *Great Britain*, or *GB*, the validator returns the value UK to the task. If the user text includes *England*, *Northern Ireland*, *Scotland*, or *Wales*, the validator returns the value that the user matches.

Regular Expression Validator

In regular expression validation, you provide a regular expression pattern that the user input string must match.

The following table describes the properties that you can set in the `regex` validation configuration object.

Property	Type	Description
`pattern`	string	(Required) A regular expression pattern in ECMAScript regular expression format. The user input text must match the pattern to be valid. The validator supports named captures, which must have the name `<return>`. The validator returns the matched value if the input is valid.
`case_ insensitive`	Boolean	(Optional) Set to **true** if you want to match values case insensitively. The default value is **false** (case sensitive matching).

For example:

```
{
   "tasks": [
      ...
   ]
   "validators": [
      {
         "id": "PRODUCT_CODE",
         "regex": {
            "pattern": "PCO-.*"
         }
      }
   ]
}
```

This validator matches any string that starts with the value `PCO-`.

Eduction Validator

In Eduction validation, you specify one or more Eduction grammars (and optionally entities) that the user input string must match. This option uses the Answer Server embedded Eduction module.

The following table describes the properties that you can set in the `eduction` validation configuration object.

Property	Type	Description
`grammars`	string	(Required) A comma-separated list of grammars to use to validate the input. You can use Wildcards in the grammar name to specify multiple grammars. However, you cannot use Wildcard values in the directory name. You can specify grammar files with an absolute path or a path relative to the Answer Server working directory.
`entities`	string	(Optional) A comma-separated list of entities in the configured grammars to use to validate the input. You can use Wildcards in the entity string to specify multiple entities.

For example:

```
{
   "tasks": [
      ...
   ]
   "validators": [
      {
         "id": "UK_PHONE",
         "eduction": {
            "grammars": "configuration/number_phone_gb.xml",
            "entities": "phone/all/gb"
         }
      }
   ]
}
```

This validator checks that the user input text contains a phone number that matches one of the types in the `phone/all/gb` entity of the `number_phone_gb.xml` grammar. If the user input text contains a valid match, the validator returns the normalized match text to the task.

Lua Validator

In Lua validation, you specify the name of a Lua function to use to validate the user input.

The function that you specify must accept a string (the text to validate) as the first parameter. You can optionally also use a `taskUtils` object as the second parameter, if you want to use `taskUtils` methods in your validator.

When the text is a valid response, the function must a string (the normalized value to use from the user input text). When the user text is not a valid response the function must either return `nil` or not return a value.

The following table describes the properties that you can set in the `lua` validation JSON configuration object.

Property	Type	Description

`function`	string	(Required) The name of the function to call. This function must exist in the Lua script that you configure in your task configuration JSON file (see Lua Processing Scripts, on page 113).

For example:

```
{
   "tasks": [
      ...
   ]
   "validators": [
      {
         "id": "FTSE_SYMBOL",
         "lua": {
           "function": "check_ftse_symbol"
         }
      }
   ]
}
```

This validator calls the `check_ftse_symbol` function in the task Lua script to validate user input text. If the user input text is valid, the validator sends the value that the function returns to the task.

Process Non-Valid Input

You can configure your response validators to call a Lua function when the user input text fails to validate.

To configure a Lua function, you set the `invalid_input_lua` property in your validator configuration to the name of the Lua function to call. This function must exist in the Lua script that you configure in your task configuration JSON file (see Lua Processing Scripts, on page 113).

You can use this function to process non-valid input further. For example, if a user asks a question instead of providing a direct answer to a requirement, you might use the `invalid_input_lua` function to send the user input text to the `Ask` action, and retrieve answers from your answer systems.

The function that you specify must take a `taskUtils` object. If the function sets a response, Answer Server returns this response to the user.

If the function does not update the response, Answer Server uses a default message *I'm sorry. I didn't understand that.*, and then repeats the requirement prompt. You can modify this message, if required (see Default Messages, on page 116).

For example:

```
{
   "tasks": [
      {
         "id" : "HOLIDAY",
         "trigger" : {
            "regex" : {
               "pattern" : "(book|go) .* holiday"
            }
```

```
         },
         "requirements": [
            {
               "id": "HOLIDAY_LOCATION",
               "prompt": "What country would you like to visit?",
               "validators": [ "COUNTRYVALIDATOR" ]
            }
         ]
      }
   ],
   "validators": [
      {
         "id": "COUNTRYVALIDATOR",
         "eduction": {
            "grammars": "configuration/place_countries.ecr"
         },
         "invalid_input_lua": "holiday_planner"
      }
   ]
}
```

If `holiday_planner` is a Lua function that uses the user input text to send an `Ask` action, and returns the answer as a prompt, you might get a conversation similar to the following (user text is in *italic*, Answer Server response in **bold**):

I'd like to go on holiday.

What country would you like to visit?

What country is Budapest in?

Hungary. What country would you like to visit?

Hungary.

Post-Task Actions

Post-task actions run at the end of the task, after all the task requirements have been satisfied. You can use these options to return an acknowledgment response to the user to confirm the details in the task, and to run a Lua function that performs some final operation with the collected task requirements.

To configure post-task actions, you set the `post` object in the configuration object for an individual task.

The following table describes the properties that you can set in the `post` object.

Property	Type	Description
`response`	string	(Optional) A string response to return to the user. For example, this might be a confirmation of the information that was collected in the task. You can include session and task variables, by inserting the variable ID in double curly brackets, for example `{{MYVARIABLE}}`. The variable must already be set in an earlier part of the task (for a task variable) or conversation session (for a session variable). You can optionally use the `task:` or `session:` prefix to

		specify the type of variable, for example `{{task:MYVARIABLE}}`. If you do not use a prefix, Answer Server searches for the variable in the task variables first, and then the session variables.
`lua`	string	(Optional) A lua function to run at the end of the task, for example to run an external process to complete the task. The function that you specify must accept a `taskUtils` object. See Lua Processing Scripts, on page 113.
`routing`	object	(Optional) A string or configuration object that determines the task to run next. See Task Routing, on the next page.

```
{
   "initial_task" : "GREET",
   "tasks" : [
      {
         "id" : "GREET",
         "pre" : {
            "response" : "Hello and welcome to the Lunch Virtual Assistant."
         },
         "requirements" : [
            {
               "id": "USER_NAME",
               "prompt": "Before we get started, could you tell me your name?",
               "scope": "session"
            }
         ],
         "post": {
            "response" : "Welcome, {{USER_NAME}}. How can we help you today?"
         }
      },
      {
         "id" : "LUNCH",
         "pre" : {
            "response" : "I can help you order some lunch."
         },
         "trigger" : {
            "regex" : {
               "pattern" : "(Book|order) .* lunch"
            }
         },
         "requirements" : [
            {
               "id" : "FOOD_TYPE",
               "prompt" : "Do you feel like a sandwich or a panini?"
            },
            {
               "id" : "FILLING",
               "prompt" : "What filling would you like?"
            }
         ],
```

```
         "post" : {
            "response" : "All right, {{USER_NAME}}. We'll get you a {{FILLING}}
{{FOOD_TYPE}} right away!",
            "lua" : "send_lunch_order",
            "routing" : "ANYTHING_ELSE"
         }
      }
   ]
}
```

In this example, the final response in the `GREET` task uses the session variable `{{USER_NAME}}`. The final response in the `LUNCH` task uses the session variable `{{USER_NAME}}`, and the task variables `{{FILLING}}` and `{{FOOD_TYPE}}`.

This simple configuration might result in a conversation like the following example:

Hello and Welcome to the Lunch Virtual Assistant.
Before we get started, could you tell me your name?

Amy

Welcome, Amy. How can we help you today?

I'd like to book lunch.

I can help you order some lunch.
Do you feel like a sandwich or a panini?

Panini

What filling would you like?

Cheese

All right, Amy. We'll get you a Cheese Panini right away!

The task then runs the `send_lunch_order` Lua function, which might call out to an external system to set up the order. It routes the user to the `ANYTHING_ELSE` task.

Task Routing

Task routing allows you to specify a task to run after the current task is complete. There are two types of routing that you can use:

- **Simple**. You provide the ID of the task to run after the current task is complete.
- **Conditional**. You specify a set of conditions (such as user selections), and specify the task to run in each case.

To use routing, you set the `routing` property in the `post` object of your task (see Post-Task Actions, on page 108).

Configure Simple Routing

In a simple routing, you set the `routing` property to the ID of the task that you want to run next.

For example:

```
{
   "initial_task" : "GREET",
   "tasks" : [
      {
         "id" : "GREET",
         "pre" : {
            "response" : "Hello and welcome to the Virtual Assistant. Before we get
started, I'd like to ask you a couple of questions."
         },
         "requirements": [
            {
               "id": "USER_NAME",
               "prompt": "What is your name?",
               "scope": "session"
            },
            {
               "id": "USER_COUNTRY",
               "prompt": "What country do you live in?",
               "scope": "session"
            }
         ],
         "post" : {
               "routing" : "CUSTOMER"
         }
      },
      ...
   ]
}
```

This example task asks the user for their name and location, and then automatically routes to the `CUSTOMER` task (which you must also define somewhere in the task configuration file).

Configure Conditional Routing

In conditional routing, the `routing` property contains a configuration object that contains details of the conditional routing.

The following table describes the properties that you can set in the `routing` configuration object.

Property	Type	Description
`prompt`	string	(Required) A prompt to send to the user to request a choice for routing. You can include session and task variables, by inserting the variable ID in double curly brackets, for example `{{MYVARIABLE}}`. The variable must already be set in an earlier part of the task (for a task variable) or conversation session (for a session variable). You can optionally use the `task:` or `session:` prefix to specify the type of variable, for example `{{task:MYVARIABLE}}`. If you do not use a prefix, Answer Server searches for the variable in the task variables first, and then the session variables.

`map`	array, object	(Required) One or more JSON objects that specify the user responses, and the routing option to use for that response. Each object contains the following properties: • `match` (string). Required. The value that triggers this option. • `routing` (string). Required. The task to route to next when the user response triggers this option. • `response` (string). Optional. A string to return to the user to confirm their option. You can include session and task variables, by inserting the variable ID in double curly brackets, for example `{{MYVARIABLE}}`. The variable must already be set in an earlier part of the task (for a task variable) or conversation session (for a session variable). You can optionally use the `task:` or `session:` prefix to specify the type of variable, for example `{{task:MYVARIABLE}}`. If you do not use a prefix, Answer Server searches for the variable in the task variables first, and then the session variables.
`validation`	array, strings	(Optional) A list of validators to use to validate the user selection. Specify the ID of each validator that you want to use to validate the response to use to route the conversation to a new task. You define the validators separately in the `validators` section of the task configuration JSON file. See Response Validation, on page 103. If you use validation, you must ensure that all the validators for a `routing` option map back to the value that you specify in your `map` object `match` properties.
`user_cancel`	object	(Optional) An object that defines keywords that a user can use to cancel a task, and the action to perform if they do. For details of the configuration properties, see User Cancellation, on page 120. The `user_cancel` options in the conditional routing section override any values that you set in the main task configuration or for the individual task.
`system_cancel`	object	(Optional) An object that defines what actions to take when the requirement receives multiple non-valid responses. For details of the configuration properties, see System Cancellation, on page 121. The `system_cancel` options in the conditional routing section override any values that you set in the main task configuration or for the individual task.

For example:

```
{
   "tasks" : [
      ...
      {
         "id" : "CUSTOMER",
         "routing" : {
            "prompt" : "Do you have an existing account with us?",
            "validation" : [ "YESNO" ],
            "map" : [
               {
```

```
                "match" : "yes",
                "routing" : "EXISTING_DETAILS",
                "response" : "Okay, {{USER_NAME}}. We'll get you the options for
existing customers."
            },
            {
                "match" : "no",
                "routing" : "NEW_OPTIONS"
            }
        ]
      }
    }
    ...
  ]
}
```

This example routes users to different tasks depending on whether they are an existing or new customer.

> **TIP:**
> When Answer Server returns a routing prompt, it also returns the valid responses (the values of the `match` property for each object in your `map` array). You can use these values to present the available options to your users, particularly if you do not want to use validation in your routing object.

Use Routing in a Lua Function

When you are using a Lua script in your task configuration, you can route to the next task in your `post` Lua function.

Answer Server calls the `post` Lua function at the end of a task, when all the requirements are met. You can use this function to run an external operation, or store data that allows an operative to complete any manual portion of the task.

You can include routing in your `post` Lua function to control the task that Answer Server occurs next. To do this, you can use the `setNextTask` method. For more information, refer to the *Answer Server Reference*.

Task routing in the `post` Lua response takes precedence over any explicit task routing in the task JSON configuration.

Lua Processing Scripts

You can use Lua functions to perform additional processing for your tasks. For example, if you have a conversation task that sets up the options that a user wants to use to open an account, or make a transaction, you might use a Lua function to call out to an external system to perform the final action.

You can create only one Lua script for your conversation system. You specify the path and file name for this script in the `lua_script` property in your task configuration JSON file. You can then call out from the configuration for an individual task to a particular function in the script file. For example:

```
{
   tasks : [
      ...
   ],
   lua_script : "C:\AnswerServer\Scripts\TaskProcessing.lua",
   validators : [
      ...
   ]
}
```

You can use a full or relative path. Relative paths can be relative to the Answer Server working directory, or relative to the directory that contains your task configuration JSON file (Answer Server checks for the file relative to the working directory first).

> **NOTE:**
> You can specify only one Lua script, even if you split your task configuration across multiple JSON files.

In your individual task configurations, you can specify the following properties to set Lua scripts:

- The `pre` object `lua` property specifies the name of a function to run before the task starts. For example, you can use this function to generate an appropriate preamble to send to the user. See Pre-Task Actions, on page 91.
- The `post` object `lua` property specifies the name of a function to run when all the task requirements have been satisfied. You can use this function to run an external process to complete the task. See Post-Task Actions, on page 108.
- The `action` object `lua` property (in the `user_cancel` or `system_cancel` objects) specifies the name of a function to run when the user or system cancels the task. See Task Cancellation, on page 120.

All `pre`, `post`, and cancel `action` Lua functions must accept a `taskUtils` object.

You can also use Lua functions in your validators, to validate user input:

- Lua validator configuration objects allow you to specify the name of a Lua function to use to validate user input. Lua validator functions must accept a string (the text to validate) as the first parameter. You can optionally also use a `taskUtils` object as the second parameter, if you want to use `taskUtils` methods in your validator. See Response Validation, on page 103.
- The `invalid_input_lua` property allows you to set the name of a function to run when the user input text does not validate. This function must accept a `taskUtils` object. See Process Non-Valid Input, on page 107.

You can use an additional Lua function to determine the message to return to a user and valid responses from the user when their input triggers multiple tasks. The Lua function must accept a `routingTable` object as the first parameter, which is can use to override the response and options. You can optionally also use a `taskUtils` object as the second parameter, if you want to use `taskUtils` methods in your function. See Default Messages, on page 116.

You can use any of the standard Lua functions and methods in your Lua functions, as well as several custom conversation system Lua methods. The conversation Lua options include simple methods to get and set task and session variables, and a method to forward a question to your other Answer Server systems.

For more information about the Lua functions and methods available, refer to the *Answer Server Reference*.

Default Tasks

In most cases, you use conversation triggers or routing to direct users to particular conversation tasks. However, you must also configure an initial task and a fallback task.

Initial Task

The initial task runs when the conversation session starts. It activates when you start a conversation by sending the `Converse` action without the `Text` parameter (see Start and Continue a Conversation, on page 126).

The main use of the initial task is to greet your users when they join a conversation session.

To define the initial task, you set the `initial_task` property in your task configuration JSON to the ID of the task that you want to use. This property is required, and the associated task must exist somewhere in your task configuration file.

For example:

```
{
   "initial_task" : "GREET",
   "tasks" : [
      {
         "id" : "GREET",
         "pre" : {
            "response" : "Hello and Welcome to the Virtual Assistant. How can I
help you?"
         }
      }
   ]
}
```

Fallback Task

The fallback task runs when there are no existing tasks in the session, for example because all previous tasks are complete.

To define the fallback task, you set the `fallback_task` property in your task configuration JSON to the ID of the task that you want to use. This property is required, and the associated task must exist somewhere in your task configuration file, even if you have explicit routing such that it is never used.

```
{
   "fallback_task" : "HELP",
   "tasks" : [
      {
         "id" : "HELP",
         "pre" : {
            "response" : "Is there anything else I can help you with today?"
```

```
         }
      }
   ]
}
```

Default Messages

The conversation system has several default messages that it uses, in addition to the responses that exist in the task configuration. These default messages have standard values in Answer Server. You can change the default values, for example, if you want to run conversations in a different language, use a different conversational style, or customize the messages to your organization.

To configure default messages, you set the `default_messages` object in your task configuration file. The following sections describe the messages that you can configure.

Response for Non-Valid Input

The non-valid input response is the value that the conversation system returns when the user input was not recognized, for example because the message did not pass validation.

To modify the default message for non-valid input, add the `default_messages` object and set the `invalid_input` property to the new message.

For example:

```
{
   "default_messages" : {
      "invalid_input" : "Je ne comprends pas."
   }
}
```

The default value for this message is:

I'm sorry. I didn't understand that.

Response for Disambiguation

The disambiguation response is the value that the conversation system returns when there are two or more possible responses. For example, if user text triggers more than one task, or if the task sends a question that has multiple interpretations to a Answer Server Fact Bank system.

To modify the default message for disambiguation, add the `ambiguous_input` property to the `default_messages` object. The `ambiguous_input` property is a configuration object.

The following table describes the properties that you can set in the `ambiguous_input` object.

Property	Type	Description
`message`	string	(Optional) The message text to display to the user. This message must include the string `[[ITEMS]]`, which is a special token that Answer Server replaces with the disambiguation items. The default value is `"Which of the following did you mean:`

		`[[ITEMS]]?"`.
`item_separator`	string	(Optional) The separator to use between the items in the disambiguation list. The default value is `", "` (that is, a comma and a space).
`last_separator`	string	(Optional) The separator to use between the second-last and last item in the list. The default value is `", or "` (that is, a serial comma and *or*, with appropriate spaces).
`lua`	string	(Optional) A Lua function to call whenever some user input triggers multiple tasks. See Lua Processing Scripts, on page 113. The Lua function must accept a `routingTable` object as the first parameter, which is can use to override the response and options. You can optionally also use a `taskUtils` object as the second parameter, if you want to use `taskUtils` methods in your function. This function can change the default prompt to return to the user, and the possible responses and tasks that they route to. The function can access the default prompts and options, as well as user input text, session variables, and so on. The function can optionally route to tasks that were not originally triggered. If the function leaves only one possible task, Answer Server routes to that task immediately without prompting the user. If the function removes all possible tasks, Answer Server behaves as if no tasks were triggered. If the function does not modify the prompt or options, Answer Server uses the default options.

For example:

```
{
   "default_messages" : {
      "ambiguous_input" : {
         "message": "Which one of [[ITEMS]] did you mean?",
         "item_separator": "/",
         "last_separator": "/"
      }
   }
}
```

This example results in a disambiguation message of the following form:

Which one of order breakfast/order lunch/order dinner did you mean?

The default configuration gives a disambiguation message of the following form:

Which of the following did you mean: order breakfast, order lunch, or order dinner?

Response for Multiple Answer Disambiguation

The multiple answer disambiguation is the message that the conversation system returns if an `Ask` action sent as part of the conversation returns multiple possible answers.

This configuration allows you to set different messages depending on the type of answer that is requested. For more information about the types of questions that Answer Server can parse, see The Question Parser Eduction Grammar, on page 69.

To modify the default message for disambiguation, add the `multiple_answers_disambiguation` property to the `default_messages` object. The `multiple_answers_disambiguation` property is a configuration object.

The following table describes the properties that you can set in the `multiple_answers_disambiguation` object.

Property	Type	Description
`entity`	object	(Optional) An object that describes the message to display when a question about a particular entity returned multiple answers. This object can contain the following properties: • `message` (string). Optional. The message text to display to the user. This message must include the string `[[ITEMS]]`, which is a special token that Answer Server replaces with the disambiguation items. It can also include the string `[[ENTITY]]`, which is a special token that Answer Server replaces with the name of the entity in the question. The default value is `"Found multiple possibilities for '[[ENTITY]]', which did you mean: [[ITEMS]]?"`. • `item_separator` (string). Optional. The separator to use between the items in the disambiguation list. The default value is `", "` (that is, a comma and a space). • `last_separator` (string). Optional. The separator to use between the second-last and last item in the list. The default value is `", or "` (that is, a serial comma and *or*, with appropriate spaces).
`qualifier_code`	object	(Optional) An object that describes the message to display when a question returned multiple answers with different qualifiers that you can use to determine the correct answer to use. This object can contain the following properties: • `message` (string). Optional. The message text to display to the user. This message must include the string `[[ITEMS]]`, which is a special token that Answer Server replaces with the disambiguation items. The default value is `"Got multiple answers. Choose a qualifier to disambiguate on: [[ITEMS]]?"`. • `item_separator` (string). Optional. The separator to use between the items in the disambiguation list. The default value is `", "` (that is, a comma and a space). • `last_separator` (string). Optional. The separator to use between the second-last and last item in the list. The default value is `", or "` (that is, a serial comma and *or*, with appropriate spaces).
`qualifier_value`	object	(Optional) An object that describes the message to display when a question returned multiple answers with different qualifiers that you can use to determine the correct answer to use. This object can contain the following

		properties: • `message` (string). Optional. The message text to display to the user. This message must include the string `[[ITEMS]]`, which is a special token that Answer Server replaces with the disambiguation items. The default value is `"Which value did you want to filter by: [[ITEMS]]?"`. • `item_separator` (string). Optional. The separator to use between the items in the disambiguation list. The default value is `", "` (that is, a comma and a space). • `last_separator` (string). Optional. The separator to use between the second-last and last item in the list. The default value is `", or "` (that is, a serial comma and *or*, with appropriate spaces).
`additional_choices`	object	(Optional) An object that describes additional messages to return . This object can contain the following properties: • `all_answers` (string). Optional. The message text to display to allow the user to choose to return all available answers. The default value is `Return all answers`. • `reject_answers` (string). Optional. The message text to display to allow the user to choose to reject all available answers. The default value is `None of these`.
`rejected_answers`	object	(Optional) An object that describes the message to display when the user rejects the available answers. This object can contain the following properties: • `message` (string). Optional. The message text to display to the user. The default value is `I'm sorry, I don't have an answer for that.`.

For example:

```
"default_messages": {
   "multiple_answers_disambiguation": {
      "entity": {
         "message": "I found several possible matches for '[[ENTITY]]'. Which of
the following matches your intent: [[ITEMS]]?"
      },
      "qualifier_code": {
         "message": "I found multiple answers for that. Pick one of these
qualifiers to clarify your original question: [[ITEMS]]"
      },
      "qualifier_value": {
         "message": "I have several answers for you, but I need more information to
give you the best one. Choose one of the following values: [[ITEMS]]"
      },
      "additional_choices": {
         "all_answers": "I want it all",
         "reject_answers": "That's not helpful"
      },
      "rejected_answers": {
```

```
         "message": "Then I'm afraid I can't help you with your question, because I
have no further information."
      }
   }
}
```

> **TIP:**
> When the `Converse` action calls out to the `Ask` action to ask a question, Answer Server returns the possible answers in the `Converse` action response, along with the disambiguation information. At this point, you can use the special string **`ANSWER_NUMBER_`*`N`*** in the `Text` parameter of the `Converse` action to return a particular answer from the list, where *`N`* is the number of the answer (starting from 0).

Task Cancellation

You can configure settings to define when and how to cancel a conversation task.

You can define task cancellation options at the top level of your task configuration file, to define global cancellation options. You can also define it in a task, a task requirement, and in a post-task conditional routing object. Values that you set in the requirement or routing objects override the settings at the task level, which override the global values.

User Cancellation

The `user_cancel` property defines settings that allows users to cancel a conversation task, and what actions to take.

The following table describes the options that you can set in the `user_cancel` object.

Property	Type	Description
`keywords`	array, strings	(Optional) A list of keywords that the user can type to cancel the task. When a user provides one of these keywords, Answer Server cancels the current conversation task, and runs the specified `action`. The default value is **cancel**. You can turn off user cancellation by setting keywords to an empty array. **NOTE:** Answer Server checks for cancellation terms before it performs response validation, so it cancels the task even if the cancellation keyword is also a valid response to the task requirement. Micro Focus recommends that you choose your response validation and cancellation keywords carefully to avoid a conflict. You can override global or task-level cancellation keywords by setting keywords in a requirement (or by setting the requirement keywords to an empty array).
`case_ insensitive`	Boolean	(Optional) Set to **false** if you want to match the specified keywords case sensitively. The default value is **true** (not case sensitive).

`action`	object	(Optional) An object that defines the action to perform when the user cancels the task. You can set the following properties: • `response` (string) Optional. The response to return when the task is canceled. You can include session and task variables. The default value is **Task canceled**. • `routing` (string) Optional. The ID of the task to route to when the task is canceled. The default value is the configured `fallback_task` (see Default Tasks, on page 115). • `lua` (string) Optional. The name of a Lua function to run when the task is canceled. See Lua Processing Scripts, on page 113. You can set this property to `null` to remove a Lua script that you configured at a more general level.

For example:

```
{
   "user_cancel" : {
      "keywords" : [ "cancel", "end", "stop" ],
      "action" : {
         "response" : "Okay {{USER_NAME}}, we'll cancel that for now.",
         "routing" : "HELP"
      }
   }
}
```

System Cancellation

The `system_cancel` property defines settings that define what actions to take when the conversation task receives multiple non-valid responses.

> **NOTE:**
> Any requirements that have `ask_options` configured ignore any system cancellation because in this case, Answer Server treats all non-valid responses as potential questions.

The following table describes the options that you can set in the `system_cancel` object.

Property	Type	Description
`max_attempts`	number	(Optional) The number of non-valid input attempts that the user can make before the system cancels the task. The default value is **0** (no limit).
`action`	object	(Optional) An object that defines the action to perform when the system cancels the task after non-valid input. You can set the following properties: • `response` (string) Optional. The response to return when the task is canceled. You can include session and task variables. The default value is **Task aborted**. • `routing` (string) Optional. The ID of the task to route to when the task is canceled. The default value is the configured `fallback_task` (see Default Tasks, on page 115).

		• `lua` (string) Optional. The name of a Lua function to run when the task is canceled. See Lua Processing Scripts, on page 113. You can set this property to `null` to remove a Lua script that you configured at a more general level.

For example:

```
{
   "system_cancel" : {
      "max_attempts" : 5,
      "action" : {
         "response" : "I'm sorry, I still didn't understand that. Would you like to
try again?",
         "lua" : "system_cancel"
      }
   }
}
```

Task Configuration Example

The Answer Server installation includes an example conversation JSON configuration and Lua script.

You can view the example files in the `conversation/examples/travel` directory in your Answer Server installation. The default Answer Server configuration file contains a commented example conversation system configuration, which you can uncomment to run the example.

Configure the Conversation Agentstore

> **NOTE:**
> The Conversation Agentstore is an optional component. You do not need to configure it if you do want to use agent triggers in your conversation tasks. See Conversation Triggers, on page 92.

The IDOL Agentstore component is a specially configured IDOL Content component.

In agent search, you send plain text or a document to the Agentstore, which returns any agents that match the document. In a Conversation system, you can store conversation triggers in the Agentstore.

The conversation task can query the Agentstore to find any conversation triggers that match the text that a user provides.

Configure the Agentstore Component

The Answer Server package includes a predefined Conversation Agentstore configuration file.

> **NOTE:**
> Micro Focus recommends that you use a separate Agentstore for Passage Extractor, Answer Bank, and Conversation answer systems.

To configure the Agentstore component for your Conversation system

1. In your Answer Server installation directory, copy the Agentstore `agentstore.cfg` configuration file.
2. Open your Agentstore installation directory.
3. Paste the Answer Server Agentstore configuration file. Overwrite the installed configuration file (you might want to make a copy of it first).
4. Open the configuration file in a text editor.
5. Update the `[License]` section with the host and port information for your License Server. For more information, see Configure the License Server Host and Port, on page 21.
6. In the `[Server]` section, find the `Port` parameter. Check that the specified port is available on the host machine, or change it to an available port.

   > **NOTE:**
   > If you modify the port, make sure to update the system configuration in your Answer Server configuration file. See Configure the Conversation System, on page 89.

7. In the `[Service]` section, find the `ServicePort` parameter. Check that the specified port is available on the host machine, or change it to an available port.
8. Save and close the configuration file.

Configure the Conversation System to Use Agentstore

To use Agent triggers, you must include the Agentstore details in your conversation system configuration by setting the `AgentstoreHost` and `AgentstoreACIPort` configuration parameters. Depending on your system configuration, you might also need to set `AgentstoreSSLConfig` and `AgentstoreGSSServiceName`.

```
[MyConversation]
Type=Conversation
TaskConfigurationFile=C:\AnswerServer\Conversation\tasks.json
// Trigger Agentstore
AgentStoreHost=localhost
AgentStoreAciport=5002
```

For more information about the available configuration parameters, refer to the *Answer Server Reference*.

Index Conversation Trigger Agents

After you have configured the Agentstore, you must index the conversation trigger agents that the Conversation system must use.

To do this, you must create the IDX or XML files that contain your conversation triggers (see Agent Triggers, on page 93). You can then use a `DREADD` index action to add the IDX or XML files to your Agentstore. For example:

```
http://localhost:5001/DREADD?C:\AnswerServer\conversation\conversation_
triggers.idx.gz
```

Chapter 10: Hold Conversations in Answer Server

This section describes how to hold conversations in Answer Server.

- Hold a Conversation 125

Hold a Conversation

To start conversations, you must create a conversation session in Answer Server, which generates a session token. You use this session token to track a particular conversation in the `Converse` action.

Create a Conversation Session

To create a conversation session, you use a `ManageResources` action in a POST request method, with the details of the operation to perform provided in the `Data` parameter as a JSON object.

```
Action=ManageResources&SystemName=MyConversations
data={
   "operation":"add",
   "type":"conversation_session"
}
```

You can optionally pass session variables to the conversation session, so that the conversation does not have to ask for information that is already available in your system. For example, you might pass in the user name from the login that the user provided when they signed into your application. For example:

```
Action=ManageResources&SystemName=MyConversations
data={
   "operation":"add",
   "type":"conversation_session",
   "session_variables": [
      {"name": "USER_NAME", "value": "Jane"},
      {"name": "COUNTRY", "value": "USA"}
   ]
}
```

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

> **NOTE:**
> The `ManageResources` action fails if you attempt to use request JSON that contains properties that are not contained in the appropriate schema.

This action returns a session token, which you can use in the `Converse` action.

Retrieve Active Conversation Sessions

To retrieve the session IDs of your active conversation sessions, you use a `GetResources` action, with `Type` set to **`conversation_session`**. For example:

```
action=GetResources&SystemName=MyConversations&Type=conversation_session
```

Start and Continue a Conversation

To run conversations, you use the `Converse` action. You must set `SystemName` to the name of the conversation system, and `SessionID` to the session token that you receive from the `ManageResources` action when you create the session.

For example:

```
action=Converse&SystemName=MyConversations&SessionID=860028728520387723
```

This example starts the conversation without providing any user text. In this case, Answer Server runs the conversation task that you have configured as your initial task and returns the response from that task. See Initial Task, on page 115.

To provide user text, you add the `Text` parameter:

```
action=Converse&SystemName=MyConversations&SessionID=8600287285203877 23&Text=I
would like to buy some shares
```

Each action with the same session ID carries on the conversation, running the appropriate conversation tasks. You provide each new user text selection in the `Text` parameter, and the action returns the response from the conversation task.

When the `Converse` action calls out to the `Ask` action to ask a question, the `Ask` response can include additional information to allow the user to disambiguate between multiple possible answers. In this case, Answer Server returns the possible answers in the `Converse` action response, along with the disambiguation information. At this point, you can use the special string **`ANSWER_NUMBER_`*`N`*** to return a particular answer from the list, where *`N`* is the number of the answer (starting from 0).

You can retrieve the XML Schema Definitions (XSDs) for the `Converse` action by using the `GetResources` action with the `Type` parameter set to **`XSD`**. For more information, refer to the *Answer Server Reference*.

Close a Conversation Session

At the end of a conversation, you can delete the conversation session to free up the session license. To delete the conversation session, you use another `ManageResources` action, with the delete operation and the appropriate session ID. For example:

```
Action=ManageResources&SystemName=MyConversations
data={
   "operation":"delete",
   "type":"conversation_session",
   "ids":["860028728520387723"]
}
```

> **TIP:**
> You can configure Answer Server to automatically close inactive conversation sessions, by setting the `SessionExpirationInterval` and `SessionExpirationIdleTime` configuration parameters in the system configuration section. For more information, see Configure the Conversation System, on page 89, and refer to the *Answer Server Reference*.

Retrieve a Conversation Transcript

You can retrieve a full transcript of a particular conversation session by using the `GetResources` action with `Type` set to **`Transcript`**, and `IDs` set to the ID of the conversation session that you want to retrieve. For example:

```
action=GetResources&Type=Transcript&SessionID=860028728520387723
```

> **TIP:**
> You can also use the transcript in your conversation Lua scripts. See Lua Processing Scripts, on page 113 and refer to the *Answer Server Reference*.

Chapter 10: Use Natural Language Generation in Answer Server

This section describes how to configure Answer Server to use natural language generation (NLG), and how to use the `NLG` action to generate sentences.

- Configure Natural Language Generation 127
- Run Natural Language Generation 128

Configure Natural Language Generation

Answer Server Natural Language Generation (NLG) generates a user-friendly natural language sentence from a JSON object that describes the parts of speech to include. You can use NLG, for example, to generate user-friendly responses for a user interface in a programmatic way.

> **NOTE:**
> You can use `NLG` only if your license permits it.

Answer Server uses the `SimpleNLG-4.4.8.jar` file to perform natural language generation. You must enable NLG and provide the location of this

To configure Answer Server to use NLG

1. Open your configuration file in a text editor.
2. Create the `[NLG]` configuration section.
3. In the `[NLG]` section, set `Enabled` to **`True`**.

4. Set `JarPath` to the path to the directory that contains the `SimpleNLG-4.4.8.jar` file. This file is available in your Answer Server installation.
5. Save and close the configuration file.

> **NOTE:**
> If you enable NLG and your license does not allow it, or Answer Server cannot find the `SimpleNLG-4.4.8.jar` in the specified directory, Answer Server does not start.

Run Natural Language Generation

You run NLG by sending the `NLG` action to Answer Server in a POST request method. You submit a JSON object, which contains details of the parts of speech that you want to use to generate a sentence, and the action returns the appropriate sentence.

```
curl http://localhost:12000/?action=NLG -F Spec={
  "sentences": [
    {
      "subject": {
        "noun_phrases": [
          { "value": "the dog" },
          { "value": "your giraffe" }
        ]
      },
      "verb": {
        "verb_phrase": {
          "value": "chase",
          "modifiers": [ "quickly" ]
        }
      },
      "object": {
        "noun_phrases": [
          { "value": "the monkey" },
          { "value": "George" },
          { "value": "Martha" }
        ],
        "conjunction": "or"
      }
    }
  ]
}
```

This action generates the following sentence:

The dog and your giraffe quickly chase the monkey, George or Martha.

You can retrieve the full schema for the JSON object to use by using the `GetResources` action. See Find the JSON Schema for Your Update, on page 41.

Part III: Appendixes

This section contains additional information about Answer Server.

- Debug Your Conversation Lua Scripts, on page 131

Appendix A: Debug Your Conversation Lua Scripts

The Answer Server installation includes IDE files to allow you to debug your Conversation system Lua scripts with the ZeroBrane Studio Lua IDE on Windows. For more information about conversation Lua scripts, see Lua Processing Scripts, on page 113.

To use Lua debugging, you must install ZeroBrane Studio (see https://studio.zerobrane.com).

After you install ZeroBrane Studio, you must:

- Copy the files from the `lua_ide_files` directory in your Answer Server installation to the ZeroBrane Studio installation directory, merging with the files that are already there,
- Copy the following files from the `lua` directory in your Answer Server installation to a debugging project directory (which can be anywhere):
  - `autn_conversation.dll` A DLL for debugging conversation Lua scripts.
  - `taskutils_factory.lua` A helper module for constructing taskUtils objects.
  - `autn_conversation.cfg` A copy of the `answerserver.cfg` to use during debugging.

You can then create a debug script to allow you to debug individual Lua functions. The Lua debug script file must have the following code at the start, where *`LuaScript`*`.lua` is your conversation Lua script file:

```
require("autn_lua_conversation").as_global() -- load lua functions into global
namespace
local factory = require("taskutils_factory") -- load factory module
require("LuaScript.lua") -- load the file that contains Lua scripts to debug
```

The rest of the debug script can call individual functions in your *`LuaScript`*`.lua`, set breakpoints and check the output.

For functions that require a `taskUtils` object, you can use the `factory.create_taskutils` helper function provided in the `taskutils_factory.lua` helper module. This helper module also lets you specify various input and output parameters, such as tables containing session and task variables, and tables for storing the result of API calls such as `setNextTask`. For more information, see the documentation in `taskutils_factory.lua`.

> **NOTE:**
> When you debug `ask` operations in your Lua scripts, the script sets up all the systems in your configuration file, but any required IDOL components and Fact Bank databases must be available.
>
> To debug Eduction validators, you must supply a License Server license key file that contains a license for Answer Server. Set the `EductionLicenseFile` parameter to the path to your license key in the conversation system configuration section of your debugging configuration file. For example:
>
> ```
> [MyConversation]
> EductionLicenseFile=C:\IDOL\licensekey.dat
> ```

Glossary

A

ACI (Autonomy Content Infrastructure)
A technology layer that automates operations on unstructured information for cross-enterprise applications. ACI enables an automated and compatible business-to-business, peer-to-peer infrastructure. The ACI allows enterprise applications to understand and process content that exists in unstructured formats, such as email, Web pages, Microsoft Office documents, and IBM Notes.

ACI Server
A server component that runs on the Autonomy Content Infrastructure (ACI).

action
A request sent to an ACI server.

Agentstore
A specialized configuration of the Content component. In an Answer Bank system, the Agentstore stores the reference questions, question equivalence classes, and answers.

answer
The response that Answer Server returns. An answer provides a precise and concise response to a reference question. Answers are included as part of a question equivalence class.

answer bank
An Answer Server system that retrieves answers from a store of reference questions and associated answers, such as an FAQ. The Answer Bank uses a specially configured IDOL Agentstore component to store the questions and answers.

answer system
A configured component of Answer Server that returns answers to questions. Systems include Answer Banks and Fact Banks. An Answer Server can have multiple systems of each type, and can retrieve answers from each configured system.

C

Content component
The IDOL component that manages the data index and performs most of the search and retrieval operations from the index.

E

entity
An object, place, or person. In a Fact Bank system, the entity is the item that the fact is about, or that the question wants to know about. For example, in the question "What is the population of the USA", USA is the entity.

F

fact bank
An Answer Server system that retrieves simple factual answers. The Fact Bank uses Eduction to parse questions, and an IDOL Content component as a fact store.

fact store
The component of a fact bank that stores the factual information for retrieval. The fact store is a specially configured IDOL Content component.

I

IDOL
The Intelligent Data Operating Layer (IDOL) Server, which integrates unstructured, semi-

structured and structured information from multiple repositories through an understanding of the content. It delivers a real-time environment in which operations across applications and content are automated.

L

License Server
License Server enables you to license and run multiple IDOL solutions. You must have a License Server on a machine with a known, static IP address.

P

property
The information associated with a particular entity. In a Fact Bank system, the property is the information that the fact provides, or that the question wants to know about. For example, in the question "What is the population of the USA", population is the property.

Q

qualifier
A term or item that modifies the property in a question. This might be a date, or an adjective that affects the meaning of a question. For example, in the question "What was the population of the USA in 1850", 1850 is a qualifier to the population property.

question equivalence class
A set of equivalent questions, which all resolve to a single reference question.

question equivalence rule
A rule that maps questions to a reference question (and therefore to a question equivalence class).

question parser
The component of a Fact Bank system that parses the incoming questions to extract entities, properties, and qualifiers. Answer Server has an embedded IDOL Eduction moedule to perform question parsing.

question set
The set of questions that can reasonably be included in the Answer Bank, regardless of whether they currently have answers.

R

reference question
The best question that a stored Answer answers. In a question equivalence class, all the equivalent questions map to the reference question.

Send documentation feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Administration Guide (Micro Focus Answer Server 11.6)

Add your feedback to the email and click **Send**.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to swpdl.idoldocsfeedback@microfocus.com.

We appreciate your feedback!