Informatica

Informatica® Address Verification (On-Premises)

6.5.0

Installation and Getting Started Guide

Informatica Address Verification (On-Premises) Installation and Getting Started Guide
6.5.0
June 2024

Publication Date: 2024-06-03

Table of Contents

Preface

Read the Informatica Address Verification (On-Premises) Installation and Getting Started Guide to learn how to install and initialize Informatica Address Verification (On-Premises).

Informatica Resources

Informatica provides you with a range of product resources through the Informatica Network and other online portals. Use the resources to get the most from your Informatica products and solutions and to learn from other Informatica users and subject matter experts.

Informatica Network

The Informatica Network is the gateway to many resources, including the Informatica Knowledge Base and Informatica Global Customer Support. To enter the Informatica Network, visit https://network.informatica.com.

As an Informatica Network member, you have the following options:

- Search the Knowledge Base for product resources.
- View product availability information.
- Create and review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to find product resources such as how-to articles, best practices, video tutorials, and answers to frequently asked questions.

To search the Knowledge Base, visit https://search.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

Use the Informatica Documentation Portal to explore an extensive library of documentation for current and recent product releases. To explore the Documentation Portal, visit https://docs.informatica.com.

If you have questions, comments, or ideas about the product documentation, contact the Informatica Documentation team at infa_documentation@informatica.com.

Informatica Product Availability Matrices

Product Availability Matrices (PAMs) indicate the versions of the operating systems, databases, and types of data sources and targets that a product release supports. You can browse the Informatica PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services and based on real-world experiences from hundreds of data management projects. Informatica Velocity represents the collective knowledge of Informatica consultants who work with organizations around the world to plan, develop, deploy, and maintain successful data management solutions.

You can find Informatica Velocity resources at http://velocity.informatica.com. If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that extend and enhance your Informatica implementations. Leverage any of the hundreds of solutions from Informatica developers and partners on the Marketplace to improve your productivity and speed up time to implementation on your projects. You can find the Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through the Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link:
https://www.informatica.com/services-and-training/customer-success-services/contact-us.html.

To find online support resources on the Informatica Network, visit https://network.informatica.com and select the eSupport option.

CHAPTER 1

Installation and Configuration

This chapter includes the following topics:

Supported Platforms

Informatica Address Verification is supported on a number of hardware and software platforms.

Address Verification is developed using the C/C++ programming language. Address Verification provides different software packages to suit the hardware and software environment in which you install Address Verification. The Address Verification software packages contain C/C++, Java, and Microsoft .NET-based APIs.

Note: You can model Address Verification implementations for other languages, such as C#, PHP, Perl, Ruby, and Python. Informatica provides technical support for the C/C++, Java, and .NET APIs. Informatica does not provide implementation-specific support.

Address Verification defines address requests and responses in JSON and defines the engine configuration in JSON.

If you call the Address Verification engine through Java, install a Java Development Kit on the machine that hosts the Address Verification engine.

You can install Address Verification on machines with the following configurations:

Operating System	Processor Architecture
Windows Server 2019	x64 (64-bit)
Windows Server 2016	x64 (64-bit)
Windows 10	x64 (64-bit)
Ubuntu 20	x64 (64-bit)
SUSE Linux Enterprise Server 15	x64 (64-bit)
RedHat Enterprise Linux 8	x64 (64-bit)

System Requirements

The system resources that Informatica Address Verification requires can vary according to your installation and your data requirements.

The machine on which you install Address Verification must have a minimum of 1 GB RAM for a C++ installation. Java and Microsoft .NET installations require additional memory.

Note: Address Verification acquires memory from the operating system and does not use Java heap memory in engine memory management.

The main address verification functionality is encapsulated in one or more *function servers*. A function server is a running instance of the Informatica data verification engine.

Each function server runs as a separate process, isolating the main process from any issues related to address processing. If a function server fails, the event is reported back to the caller as an error, the function server process is terminated, and a new function server is started.

Each function server, including standby function servers, require 455 MB of RAM for processing and data structure operations. If you enable hot swapping, the total memory usage doubles for Address Verification.

For each job, you need a variable amount of memory based on your configuration settings. For more information on memory configuration for function servers, see "IDVEInformation Elements in the IDVEState.schema.json file" on page 20.

The `IDVE.h` header file contains the functions that you use to define and run the address jobs.

Before you finalize the memory requirements, consider the size of the reference address databases that you require. The complete set of worldwide postal reference databases, including supplementary databases for address enrichments, consumes approximately 55 GB of storage space.

Preloading databases into memory significantly improves the performance of Address Verification. The machine on which you install Address Verification must have sufficient RAM to preload the databases that you require.

As the total size of the worldwide databases is approximately 55 GB, the RAM required to preload all databases and perform address processing is approximately 60 GB.

For smaller file sizes, less RAM is required. For example, the worldwide batch and interactive databases consume approximately 8 GB of space, and the geocoding databases consume an additional 4.5 GB. To

preload and run the worldwide batch, interactive, and geocoding databases, the maximum amount of RAM that you require is 12 GB. The enrichment databases require 2.3 GB RAM and the quick capture databases require 33.1 GB RAM.

Tip: If fully preloading databases is not an option, use solid-state drives to store the reference address databases. Solid-state drives are faster than hard-disk drives and can significantly improve performance, especially when multithreading is used.

You set the database preloading method in the `IDVEConfig.json` file. For information about the database preloading, see "FunctionServers Elements in IDVEConfig.json" on page 18.

Memory Requirements

Informatica Address Verification stores different types of objects, such as address objects, pre-loaded reference address databases, and caches, in memory. When you define memory allocations for Address Verification, consider the different objects and their combined memory requirements.

The IDVEConfig.json file stores the properties that define the memory requirements. Update the property values if necessary to suit your installation.

Consider the following values in the IDVEConfig.json file when you review the memory requirements:

- MaxMemoryMegabytes. The total amount of memory that the data verification engine may allocate for data preloading and processing.
- NumFunctionServers. The number of function servers available for processing. Also, determines the number of inputs that the engine can process in parallel.
- NumHotStandbyFunctionServers. The number of standby function servers.
- FileSetsHotSwappingEnabled. Enables or disables dynamic switches between two file sets.
- MaxNumJobs. The maximum number of jobs that can exist concurrently.
- MaxNumInputs. The maximum number of addresses that you can submit for verification in a single job.
- MaxNumResults. The maximum number of address results that the function can return after execution.
- MaxNumVariants. The maximum number of output variants that the function can return after execution.
- NumLargeInternalOutputBuffers. The number of large output buffers that the engine creates during initialization. The buffers contribute to the required memory when you use the Java or Microsoft .NET wrapper that Informatica provides for the engine.

Rules and Guidelines for Memory Requirements

Consider the following rules and guidelines for when you define the memory for Address Verification:

- The default configuration values on the IDVEConfig.json file are sufficient to initialize the engine, but they are not likely to be adequate for data preloading.

  The default value on the MaxMemoryMegabytes property is 1024 MB.
- Engine execution takes place outside Java in unmanaged code.
- The Java and .NET APIs are wrappers for the C API.

Calculating Memory Requirements

You can use the following formula to calculate the amount of shared memory that the engine will try to allocate during initialization:

```
34 MB + (MaxNumJobs * Job Size in MB) + (Total Function Server Count * 35 MB)
```

The formula contains the following elements:

- 34 MB and 35 MB. Constant values that represent internal engine activity.
- MaxNumJobs. The maximum number of jobs that can exist concurrently, as represented by the MaxNumJobs property.
- Total Function Server Count. The sum of the NumFunctionServers and NumHotStandbyFunctionServers property values. If you enable file set hot-swapping, double the values.
- Job Size. The job size recorded in the initialization log. The following string shows a sample log entry:

  ```
  [IDVE] STATUS: Creating AV jobs in shared memory (16 jobs at 54608085 B each)
  ```

  In this string, 54608085 bytes represents the job size.

Sample Scenarios

The following table describes a range of installation scenarios and the approximate quantities of memory that each may require:

	Batch, single input, medium volumes (10M+)	**Batch, single input, large volumes (100M+)**	**Batch, bulk input, large volumes (100M+)**	**QuickCapture / Interactive, large volumes (100M+)**	**Enterprise service provider, extra large volumes (500M+)**
NumFunctionServers	4	8	8	8	16
NumHotStandby FunctionServers	1	1	1	1	4
FileSetsHotSwapping Enabled	false	false	false	false	true
MaxNumJobs	4	8	8	8	16
MaxNumInputs	1	1	1000	1	100
MaxNumResults	1	1	1	100	100
MaxNumVariants	1	3	3	3	3
Minimum required memory (approximate)	**2,592 MB**	**4,612 MB**	**8,772 MB**	**4,967 MB**	**20,975 MB**
Shared memory portion (approximate)	211 MB	355 MB	4,503 MB	708 MB	2,268 MB

Note: The minimum required memory value in each scenarios does not include memory for data preloading.

Installing Address Verification

You download the installation package from Informatica and extract the package contents to the machine on which you'll run Informatica Address Verification.

Package Contents

When you extract the Informatica Address Verification software package, you create the following directories:

- `bin`. Contains the executable file for function servers and the sample application binaries.

  The folder contains the following sample applications:

  - IDVEConsoleSample. The native C/C++ sample application.

  - IDVEConsoleSampleJava. The Java sample application.

  - IDVEConsoleSampleNET. The .NET sample application.

- `etc`. Contains the JSON configuration file.
- `include`. Contains the C/C++ library header file.
- `lib`. Contains C/C++, Java, and .NET library files.
- `src`. Contains the sample application source code.
- `doc`. Contains the JSON schema files.

  Note: All parameters, elements, and values in the Address Verification schemas are case-sensitive.

Before You Install

Consider the following rules and guidelines before you install Address Verification:

- Do not alter the directory structure of the installation package that you extract.
- If you plan to retain an older version of Address Verification on the same machine, ensure that you extract and install the current version of Address Verification in a different location.

Installing the Informatica Address Verification C/C++-based Package

To install the C/C++-based package, complete the following steps:

1. Extract the installation directories and files from the Informatica Address Verification software package.
2. Based on the platform on which you install Address Verification, copy the `.dll` or `.so` file to a shared library path on your machine.

   To find a shared library path on a Windows machine, run the `echo %path%` command from the command prompt.

   To find a shared library path on a UNIX machine, run the `echo $LD_LIBRARY_PATH` command from the command prompt.

Installing the Informatica Address Verification Java-based Package

To use the Java-based version of Informatica Address Verification, you must install a Java Development Kit on the machine that hosts the Address Verification engine. If you want to develop your own applications, you must install a Java platform on the machine.

Windows Installation

To install the Java-based package on a Windows machine, complete the following steps:

1. Extract the installation directories and files from the Informatica Address Verification software package.
2. Copy `IDVE.dll` and `IDVE.jar` to the JRE class path.

   Typically, `C:\Program Files\Java\jre\lib\ext` is saved to the system-wide class path.

   You can explicitly set application-specific class paths with the `-cp` switch.

UNIX Installation

To install the Java-based package on a UNIX machine, complete the following steps:

1. Extract the installation directories and files from the Informatica Address Verification software package.
2. Copy `IDVE.jar` and `IDVE.so` to the JRE class path.

   Typically, `/usr/j2se/jre/lib/ext` is saved to the system-wide class path.

   You can explicitly set application-specific class paths with the `-cp` switch.

Installing the Informatica Address Verification Microsoft .NET-based Package

The Windows package for .NET contains `lib/IDVE_NET.dll`, which is the .NET binding for IDVE, and `doc/IDVE_NET.xml`, which is the API documentation for the .NET DLL.

To install the .NET-based-package on a Windows machine, complete the following tasks:

1. Extract the installation directories and files from the Informatica Address Verification software package.
2. Add a reference to the `IDVE_NET.dll` assembly to your .NET project

   Note: The `IDVE_NET.dll` file requires that the native `IDVE.dll` file is present in the regular .NET DllImport search paths.

Configuring the Engine

Informatica Address Verification uses the Intelligent Data Verification Engine (IDVE) to process the verification jobs that you define.

The `IDVEConfig.json` file provides configuration information for Address Verification engine operations. Before initializing the engine, verify or configure the elements and properties in the file.

The elements and properties in the `IDVEConfig.json` file are organized in the following groups:

- Core. For information on elements and properties in the Core group, see "Core Elements in IDVEConfig.json" on page 13.
- Functions. For information on elements and properties in the Functions group, see "Functions Elements in IDVEConfig.json" on page 18.
- FunctionServers. For information on elements and properties in the FunctionServers group, see "FunctionServers Elements in IDVEConfig.json" on page 18.

Note: The elements and properties in the `IDVEConfig.json` file also appear in the Config group in the `IDVEState.json` file.

The `IDVE_GetJSON` file provides

Core Elements in IDVEConfig.json

The Core elements in the `IDVEConfig.json` file are organized under System elements, InitializationLog elements, and ErrorLog elements.

System Elements

Find the following elements under `Core/System`:

NumFunctionServers

Specifies the number of function servers available for processing. Also, determines the number of inputs that the engine can process in parallel.

NumHotStandbyFunctionServers

Specifies the number of standby function servers.

FileSetsHotSwappingEnabled

Enables or disables dynamic switches between two filesets. Supported values are true and false. Default is true.

Note: When enabled, hot swapping increases the maximum memory requirements.

MaxMemoryMegabytes

Specifies the maximum amount of memory available for all processes, including file preloading. The default value is 1024 MB.

Note: 1024 MB is a minimum value. Depending on your workload, you might need to set a higher value.

FunctionServerInitTimeoutSeconds

Specifies the timeout period to successful initialize all function servers. Default is 600 seconds.

FunctionServerCallTimeoutSeconds

Specifies the timeout period to successfully submit a job call to a function server. Default is 60 seconds.

LicenseFilesDirectoryPath

Specifies the directory path to the license files.

Note: Do not rename any license key file.

FileSetsDirectoryPath

Specifies the directory path to the reference data filesets and to the `FileSetsInfo.json` file.

The `FileSetsInfo.json` file describes the current status of the filesets. Address Verification creates the `FileSetsInfo.json` file in the directory.

Address Verification reads the `FileSetsInfo.json` file each time you initialize the engine and saves the file at each initialization. Address Verification also reads and updates the file during hot-swap operations.

Note: If the directory that the element specifies does not permit updates to the `FileSetsInfo.json` file, copy the file to a writable location. Use the `OverrideFileSetsInfoFilePath` element to identify the location of the writable file. Do not delete the original file.

OverrideFileSetsInfoFilePath

Specifies the directory path to an alternative version of the `FileSetsInfo.json` file. Create an alternative version of the file when the `FileSetsInfo.json` file that Address Verification creates by default resides in a read-only directory. When you add a path to the `OverrideFileSetsInfoFilePath` element, Address Verification reads and updates the alternative version of the file.

ExecutablesDirectoryPath

Specifies the directory path to the FunctionServer executable application.

Do not include the function server executable file name in the path.

DeleteOutdatedDataFilesAtInit

Deletes any reference data file from the `FileSetsDirectoryPath` directory that meets the following criteria:

- A newer version of the file is present in the `FileSetsDirectoryPath` directory.
- The newer file version is compatible with the current engine.

Default value is false. To activate the property, set the value to true.

The engine reviews the directory contents and deletes outdated files during initialization.

The engine uses the minimum engine version number in the file name to determine the age and compatibility of each file. For example, the file US_ADV_VRF_QCP_001_6_2_0.MD6 is compatible with version 6.2.0 and later engine versions. A reference data file will be incompatible with the engine if the engine version that the file name specifies is newer than the installed engine.

Do not enable `DeleteOutdatedDataFilesAtInit` if different engine versions read the `FileSetsDirectoryPath` directory. A given file may be compatible with one engine version but not with another.

Note: In cases where the engine deletes an outdated file, it will not return an 11000 warning to indicate that the file was skipped in preference for a newer file. Instead, the engine logs an INFO-level message to indicate that it deleted the file.

Bindings

Specifies how the data verification engine uses memory when you use the Java or Microsoft. NET wrapper that Informatica provides for the engine. Use the NumLargeInternalOutputBuffers property to define the memory policy.

The NumLargeInternalOutputBuffers property specifies the number of large output buffers that the engine creates during initialization. Each output buffer is large enough for a full AVJob JSON document. The memory used by large output buffers is deducted from the total amount that the MaxMemoryMegabytes property defines. Without a large output buffer, function calls run on a relatively small amount of stack memory. If the available memory is insufficient to return all job data, the engine returns an IDVE_SC_ERR_OUTPUT_BUFFER_TOO_SMALL error code.

By default, the engine creates a single large output buffer. If you prefer to save memory, for example, if you can perform small portions of a job at a time, you can set the NumLargeInternalOutputBuffers value to zero.

The number of large output buffers can limit the number of concurrent function calls that the engine can run if the calls require a large amount of data, as each buffer is locked for the duration of the call. If all buffers are in use, any additional calls are queued. Specify multiple buffers if you intend to run multiple large jobs in separate threads.

InitializationLog Elements

Find the following elements under `Core/InitializationLog`:

FilePath

Specifies the path to the initialization log file and the log file name.

LogLevel

Specifies the minimum severity that will be logged to the initialization log file. Any event with a lower severity does not appear in the log file. The default value is Warning.

The following list identifies the values that you can set in descending order from the most severe to the least severe.

- VeryCriticalError
- CriticalError
- Error
- Warning
- Info
- Status

ClearAtInit

Specifies whether to clear all initialization log file contents during initialization. Supported values are true and false. The log file contents are cleared during initialization when the element is set to true. Default is true.

History

Defines the policy that the engine uses to create and manage initialization log files. The engine reads the following properties to define the log file policy:

- FileSizeSoftLimitMegabytes. Specifies an optional maximum size for an initialization log file. The default value is 0, which indicates that there is no limit to the file size. If you specify a maximum file size, the verification engine reads the FileCount value and performs one of the following actions:
  - The engine creates a new file and stores the previous file as history on the local disk.
  - The engine clears the earlier contents of the current file and reuses the current file.

  If the addition of a log entry breaches the file size limit, the verification engine saves the complete log entry to the current file. The engine then creates a new file or erases the earlier file contents as indicted by the FileCount value.

  If you accept the default FileSizeSoftLimitMegabytes value, the engine ignores the FileCount value.
- FileCount. Specifies the maximum number of log files that the verification engine stores on the local disk in addition to the current log file. The default number is 0, which indicates that the engine does not store any log files on the local disk other than the current file. The range is from 0 through 999.

  Note: You can use the FileSizeSoftLimitMegabytes value and the FileCount value to define a policy for log file creation. For example, if you set the FileSizeSoftLimitMegabytes value to 10 and accept the default FileCount value, the engine writes log information to a single file and erases all older log information when the file size reaches 10 MB.
- FileNameNumberPosition. Specifies the position of the log file number in the log file name. You can set the following values:
  - InsertBeforeExtension. Adds the file number before the file extension, for example, IDVE_ErrorLog.001.txt.
  - AppendAtEnd. Adds the file number after the file extension, for example, IDVE_ErrorLog.txt.001. Append at End is the default value.

  Note: If you change the *FileNameNumberPosition* property from one value to another when the *ClearAtInit* property is set to *true*, the engine does not delete any log file named in the old format when it reinitializes.

ErrorLog Elements

Find the following elements under `Core/ErrorLog`:

FilePath

Specifies the path to the error log file and the log file name.

ClearAtInit

Specifies whether to clear all error log file contents during initialization. Supported values are true and false. The log file contents are cleared during initialization when the element is set to true. Default is true.

History

Defines the policy that the engine uses to create and manage error log files. The engine reads the following properties to define the log file policy:

- FileSizeSoftLimitMegabytes. Specifies an optional maximum size for an error log file. The default value is 0, which indicates that there is no limit to the file size. If you specify a maximum file size, the verification engine reads the FileCount value and performs one of the following actions:
  - The engine creates a new file and stores the previous file as history on the local disk.
  - The engine clears the earlier contents of the current file and reuses the current file.

  If the addition of a log entry breaches the file size limit, the verification engine saves the complete log entry to the current file. The engine then creates a new file or erases the earlier file contents as indicted by the FileCount value.

  If you accept the default FileSizeSoftLimitMegabytes value, the engine ignores the FileCount value.
- FileCount. Specifies the maximum number of log files that the verification engine stores on the local disk in addition to the current log file. The default number is 0, which indicates that the engine does not store any log files on the local disk other than the current file. The range is from 0 through 999.

  Note: You can use the FileSizeSoftLimitMegabytes value and the FileCount value to define a policy for log file creation. For example, if you set the FileSizeSoftLimitMegabytes value to 10 and accept the default FileCount value, the engine writes log information to a single file and erases all older log information when the file size reaches 10 MB.
- FileNameNumberPosition. Specifies the position of the log file number in the log file name. You can set the following values:
  - InsertBeforeExtension. Adds the file number before the file extension, for example, IDVE_ErrorLog.001.txt.
  - AppendAtEnd. Adds the file number after the file extension, for example, IDVE_ErrorLog.txt.001. Append at End is the default value.

  Note: If you change the *FileNameNumberPosition* property from one value to another when the *ClearAtInit* property is set to *true*, the engine does not delete any log file named in the old format when it reinitializes.

LogInput

Specifies the quantity of input data that the error log file stores for each log entry.

You can set the following values:

- CompleteInput. The error log contains complete data for the input that caused the issue.
- OnlySourceID. The error log contains the source ID value of the input that caused the issue. No further data is stored.
- Disabled. The error log contains an entry to state that the function server encountered an issue. No further data is stored. Default is Disabled.

InputJSONFormat

Defines the format of the data that the engine writes to the error log.

The engine uses a common set of options to configure the error log format and to configure the format of the response that it returns when you submit a function call. For more information about formatting options for the response, see "Formatting the JSON Response" on page 26

The following table defines the options that you can set:

Option	Values	Description
Format	Condensed	Removes all white space, such as character spaces, tabs, and line breaks.
	Structured	Formats objects and arrays with line breaks and indentations.
	Smart	Organizes smaller objects and arrays on a single line if the MaxLineLength value permits. Otherwise, organizes smaller objects and arrays with fewer line breaks. Default is Smart.
BOM	true/false	Prepends the output string with UTF-8/UTF-16 BOM, depending on the API call. Default is false.
HideDefault	true/false	Omits properties whose values are equal to their schema defaults. Default is false in the data response and true in the error log output.
OpenBraceOnSameLine	true/false	Places opening braces and brackets on the same line as the object or array name instead of the line below. Applies when the Format value is Structured or Smart. Default is true.
Indent	Tab	Specifies that each indentation level will consist of a single tab character ("\t").
	Spaces	Specifies that each indentation level will consist of a number of space characters (" "). See also IndentWidth. Default is Spaces.
IndentWidth	<integer>	Specifies the number of character spaces to use at each indentation level. Applies when the Indent value is Spaces. Default is 2.
MaxLineLength	<integer>	Line length value to use in calculation of dynamic line breaks. Applies when the Format value is Smart. Not a hard limit. Lines may be longer in deeply nested documents. Default is 80.
NewLine	Lf/CrLf (default Lf)	Specifies the character sequence for new lines. The NewLine option can either be carriage-return and line-feed (CrLf), or only line-feed (Lf). **Note:** The NewLine option is not present in the IDVEConfig.json file. You can add a NewLine value as an option in a URI query string.
CompactEscape	true/false (default true)	Specifies the use of compact escape sequences such as "\t" instead of "\u0009" where possible. Default is true.
EscapeSolidus	true/false (default false)	Specifies the use of escaped forward slashes. For example, "/" becomes "\/" or "\u002F" based on the CompactEscape option.

Functions Elements in IDVEConfig.json

The Functions elements in the `IDVEConfig.json` file are organized under AddressVerification elements.

AddressVerification Elements

Find the following elements under `Functions/AddressVerification` :

MaxNumJobs

Specifies the maximum number of jobs that can exist concurrently. Default is 1.

MaxNumInputs

Specifies the maximum number of addresses that you can submit for verification. The range is from 1 through 1000. Default is 100.

MaxNumResults

Defines an upper limit for the number of address suggestions that the engine can process. The default value is 20.

The engine reads the MaxNumResults property during initialization. To change the MaxNumResults property value, first deinitialize the engine.

The maximum value that you can set on the MaxNumResults property is 100. Therefore, the maximum value that you can set on the DefaultMaxNumResults property or on the MaxResultCount property in an address job is also 100.

MaxNumVariants

Specifies the maximum number of output variants that the function server can return for an address job. The range is from 1 through 10. Default is 1.

ListDataFilesUsedForOutput

Determines the data set that the engine uses during address validation and enrichment for multiple addresses at a time. The default value is false.

If you set the ListDataFilesUsedForOutput value to true, you will see the following database details in each address output:

- Country ISO 3
- Database Type
- Database SubType
- Data Set

DefaultMaxNumResults

Defines a standard value for the maximum number of address suggestions that a job that can return. If you do not set a value on the MaxResultCount property when you define a job request, the job reads the DefaultMaxNumResults property.

By default, the DefaultMaxNumResults property uses the value on the MaxNumResults property. Do not set a DefaultMaxNumResults property value that exceeds the MaxNumResults value.

FunctionServers Elements in IDVEConfig.json

The FunctionServers elements in the `IDVEConfig.json` file are organized under FileSets elements.

FileSets Elements

Find the following elements under `FunctionServers/FileSets`:

PreloadingMethod

Specifies how Address Verification preloads the reference address databases to memory. To optimize performance, you can preload reference address databases.

Set the following value:

- Map. Address Verification uses the file mapping mechanism of the operating system. Map is the default and recommended option.

PreloadingPriorityList

Specifies the order in which Address Verification loads data files into memory. Address Verification loads the database files in the order in which they appear in the list. Each preloaded file consumes a quantity of the available memory. If the maximum memory threshold is reached, further file data will not be preloaded to memory but will remain available for processing.

Includes the following elements:

- Selector. Identifies the types of file data to prioritize during the preload operation.

  The Selector is a string that specifies a Function, Country, Process Type, Process Sub Type, and DataSet ID. For example, the following values for Selector instruct Address Verification to load all data files for batch and interactive verification for Germany in the 000 data set:

  ```
  AddressVerification/DEU/Verify/BatchInteractive/000
  ```

  Note: Replace any value between a [/] separator with an asterisk to select all options at that position. For example, the following parameters specify batch and interactive data files for all countries in the 000 data set:

  ```
  AddressVerification/*/Verify/BatchInteractive/000
  ```

  If you use multiple trailing asterisks in the Selector parameter, you can replace them with a single asterisk. For example, `AddressVerification/*/*/*/*` and `AddressVerification/*` have the same meaning.

- PreloadingExtent. Specifies the preloading type for each of the reference address databases.

  Set one of the following values:

  - Full. Address Verification copies the entire reference address database to memory.
  - Partial. Address Verification loads fragments of each data file, starting with the most important data.
  - None. Address Verification does not preload any data.

Address Verification can stop preloading data at any time if the maximum memory threshold is reached. The unloaded reference data remain in the system storage.

Retrieving the State of the Engine

To retrieve the current state of the data verification engine, use an IDVE_GetJSON() or IDVE_GetJSONW() function call and pass the URI value in the function call as *State*. The IDVE_GetJSON() function returns the data in a UTF-8-encoded string, while the IDVE_GetJSONW() function returns the data in a UTF-16-encoded string.

The following sample code shows an IDVE_GetJSON() call that can retrieve the engine state:

```
char sJSON[ 50*1024];
        char sExtStatusMsg[ IDVE_EXT_STATUS_MSG_BUFFER_SIZE];
        IDVE_StatusCode i32StatusCode= IDVE_GetJSON("", "State", sJSON, sizeo
        f(sJSON), NULL, NULL, sExtStatusMsg);
```

The `IDVEState.schema.json` file lists the elements and properties that each function call returns.

The elements and properties in the file are organized in the following groups:

- DateTime. For information on elements and properties in the DateTime group, see "DateTime Element in the IDVEState.schema.json file" on page 20
- SystemInformation
- IDVEInformation
- Config
- Licenses
- Functions
- FileSets

DateTime Element in the IDVEState.schema.json file

The DateTime element is a string that contains the date and time at which you called the IDVE_GetJSON() or IDVE_GetJSONW() functions to retrieve the engine state.

SystemInformation Elements in the IDVEState.schema.json file

The SystemInformation elements provide general information about the environment in which the data verification engine is installed.

Find the following elements in the SystemInformation group:

Platform

Identifies the name and version number of the operating system on which the engine is installed.

JavaVersion

Identifies the version of Java that the engine can access.

JavaVMArguments

Identifies any Java argument applicable to the Java virtual machine.

DotNetInfo

Identifies the version of Microsoft .NET that the engine can access.

SystemMemoryMegabytes

Identifies the amount of installed physical memory on the host system.

IDVEInformation Elements in the IDVEState.schema.json file

The IDVEInformation elements describe the startup configuration of the engine.

Find the following elements in the IDVEInformation group:

Version

Identifies the data verification engine framework version.

TargetPlatform

Identifies the name of the operating system for which Informatica built the engine.

TotalMemoryUsageMegabytes

Identifies the maximum amount of memory available for all processes, including shared memory and memory allocated to data preloading.

Note: This value differs from the `MaxMemoryMegabytes` property in the `Config` group. The `MaxMemoryMegabytes` property defines the memory limit at which the engine does not initialize.

FunctionServerMaxMemoryMegabytes

Identifies the maximum amount of memory available to each function server for all processes, including job processing and data preloading.

PerFunctionServerPreloadingCapacityMegabytes

Identifies the amount of memory available to each function server to preload data.

Subtract this value from the `FunctionServerMaxMemoryMegabytes` value to find the amount of memory dedicated to each function server for processing and data structure operations.

TotalPreloadingCapacityMegabytes

Identifies the total amount of memory available for preloading data. If hot-swapping is disabled, this value is identical to the PerFunctionServerPreloadingCapacityMegabytes value.

TotalPreloadingUsageMegabytes

Identifies the total amount of memory that is currently used for data preloading.

InitializationLog

Identifies the status of the initialization log.

The log can have one of the following status values:

- Enabled. The engine configuration enabled the initialization log and the log is operational.
- DisabledInConfig. The initialization log is disabled in the engine configuration.
- DisabledBecauseOfError. The engine configuration enabled the initialization log, but the log is disabled due an error during runtime.

A status value of DisabledBecauseOfError triggers the population of two additional properties, `ErrorCode` and `ErrorMessage`, that describe the error that disabled the log.

ErrorLog

Identifies the status of the error log.

The log can have one of the following status values:

- Enabled. The engine configuration enabled the error log and the log is operational.
- DisabledInConfig. The error log is disabled in the engine configuration.
- DisabledBecauseOfError. The engine configuration enabled the error log, but the log is disabled due an error during runtime.

A status value of DisabledBecauseOfError triggers the population of two additional properties, `ErrorCode` and `ErrorMessage`, that describe the error that disabled the log.

Config Elements in the IDVEState.schema.json file

The elements and properties in the Config group provide configuration information for engine operations. They replicate the elements and properties in the `IDVEConfig.json` file.

The elements and properties in the group are further organized in the following groups:

- Core. For information on elements and properties in the Core group, see "Core Elements in IDVEConfig.json" on page 13.
- Functions. For information on elements and properties in the Functions group, see "Functions Elements in IDVEConfig.json" on page 18.
- FunctionServers. For information on elements and properties in the FunctionServers group, see "FunctionServers Elements in IDVEConfig.json" on page 18.

Licenses Elements in the IDVEState.schema.json file

The Licenses elements and properties in `IDVEState.schema.json` specify an array of licenses that IDVE loads. Each license file has a .LIC extension.

SchemaVersion

Contains a set of properties that identify the version of the schema in which the license files are written.

Find the following elements under `Licenses/SchemaVersion`:

- Major
- Minor
- Descriptive
- Revision

The property values collectively constitute a single string that identifies the version of the schema. For example, the version number 1.0.0.0 includes individual values from the Major, Minor, Descriptive, and Revision properties.

General

Find the following elements under `Licenses/General`:

CustomerID

Identifies the Customer ID for the user who has the license.

LicenseID

Identifies the identity of the license.

IssueDate

Identifies the date of issue of the license.

StartDate

Identifies the beginning of the validity period of the license.

Note: The license does not activate any functionality before the start date.

ExpirationDate

Identifies the end of the validity period of the license.

Note: The license does not activate any functionality after the end date.

ExecutionContexts

Contains properties that indicate the run-time environments in which the engine can operate.

ExecutionContexts contains the following properties:

OS_Architectures

Identifies the operating systems on which the engine can run.

Environment

Identifies any additional restrictions on the run-time environments in which the engine can operate.

MaxNumCores

Identifies and limits the maximum number of active function servers.

Note: If the number of function servers in the configuration exceeds limit, then the other function servers get into hot-standby mode and are can no longer process resources.

DeploymentType

Identifies whether the issued license is for development and testing or for production.

AddressVerification

Find the following elements under `Licenses/AddressVerification`:

MinVersion

Identifies the minimum required version of Address Verification for valid license.

Note: The *AddressVerification* part of the license does not have any effect without the Address Verification version.

GeneralFeatures

Contains properties that indicate the depth of information that the engine can return in an address verification job.

GeneralFeatures contains the following properties:

VerificationLevel

Identifies the verification level that currently limits any address verification job that you run in the engine.

DetailedStatus

Indicates whether you can request element status values in an address verification job.

MultipleVariants

Indicates whether you can request more than one result variant in an address verification job.

SubItems

Indicates whether you can request sub-items in an address verification job.

CountrySpecificFeatures

Identifies the reference database files that current licenses permit the engine to read.

CountrySpecificFeatures contains the following properties:

ProcessType

Identifies the types of data processing operation for which the engine is eligible to read reference database files, based on the current licenses. `ProcessType` is a string property. For example, a value of Verify indicates that the engine can verify addresses.

ProcessSubTypes

Identifies the process modes in which the engine is eligible to read reference database files, based on the current licenses. For example, a value of `Batch` indicates that the engine can run in batch mode. The `ProcessSubTypes` property can return multiple process modes in an array.

Countries

Identifies the countries for which the engine is eligible to read reference database files, based on the current licenses. The file identifies the countries by ISO code. For example, a value of DEU indicates that the engine can process data from Germany. The `Countries` property can return multiple ISO codes in an array.

DataSets

Contains the three-digit IDs of the types of reference data set that the engine can read, based on the current licenses. The `DataSets` property can return multiple IDs in an array.

The properties under *CountrySpecificFeatures* apply collectively to define the reference database files that the engine can read. For example, a `ProcessSubTypes` value of `Batch` does not indicate that the engine can read any file of batch data, and a Countries value of DEU does not indicate that the engine can read any file of German data.

Functions Elements in the IDVEState.schema.json file

The Functions elements in the `IDVEState.schema.json` file are organized under AddressVerification elements.

AddressVerification

Find the following elements under `Functions/AddressVerification`:

Version

Identifies the version number of the engine that the IDVEState or schema or json file applies to.

AMASVersion

Identifies the most recent release of Address Verification to receive AMAS certification at the time that your engine was released.

CASSVersion

Identifies the most recent release of Address Verification to receive CASS certification at the time that your engine was released.

SERPVersion

Identifies the most recent release of Address Verification to receive SERP certification at the time that your engine was released.

SendRightVersion

Identifies the most recent release of Address Verification to receive SendRight certification at the time that your engine was released.

SNAVersion

Identifies the most recent release of Address Verification to receive SNA certification at the time that your engine was released.

FileSets Elements in the IDVEState.schema.json file

The FileSets elements provide information about the reference datasets that the engine opens. The `IDVEConfig.json` file helps you determine the datasets that the engine can open.

Find the following elements in the FileSets group:

FileSetsInfo

Identifies the current status of the file sets. The status is the same as that in the `FileSetsInfo.json` on disk.

FileSetsInfo contains the following properties:

FileSetAState

Identifies the state of FileSetA. The property can have one of the following values:

- InUse. Indicates that FileSetA is currently loaded by the engine.
- Unused. Indicates that no files from FileSetA are loaded.
- ReadyToUse: Indicates that a user has triggered a hot-swap operation through a manual update to the `FileSetsInfo.json` file and that the engine is about to load FileSetA. To trigger the hot-swap operation, update the FileSetA property in the `FileSetsInfo.json` file from Unused to ReadyToUse. Alternatively, you can trigger a hot-swap through an API call.

  Note: If you use an API call to trigger the hot swap, the state value changes from Unused to PreparingForUse and does not record a value of ReadyToUse.
- PreparingForUse: Indicates that the engine is loading the files in FileSetA. Once the files are loaded, the property is set to InUse.
- StillInUse: Indicates that the engine has switched from FileSetA to FileSetB in a hot-swap operation, but one or more address jobs that began on FileSetA have yet to complete.

FileSetBState

Identifies the state of FileSetB. The property can have one of the following values:

- InUse. Indicates that FileSetB is currently loaded by the engine.
- Unused. Indicates that no files from FileSetB are loaded.
- ReadyToUse: Indicates that a user has triggered a hot-swap operation through a manual update to the `FileSetsInfo.json` file and that the engine is about to load FileSetB. To trigger the hot-swap operation, update the FileSetB property in the `FileSetsInfo.json` file from Unused to ReadyToUse. Alternatively, you can trigger a hot-swap through an API call.

  Note: If you use an API call to trigger the hot swap, the state value changes from Unused to PreparingForUse and does not record a value of ReadyToUse.
- PreparingForUse: Indicates that the engine is loading the files in FileSetB. Once the files are loaded, the property is set to InUse.
- StillInUse: Indicates that the engine has switched from FileSetB to FileSetA in a hot-swap operation, but one or more address jobs that began on FileSetB have yet to complete.

FileSetA

Identifies the data files in the FileSetA folder. Each file has a .MD6 extension, and provides the following information:

- FilePath. Identifies the path to the data file on the disk.
- Status. Identifies the status of the file. The status can be Active or NotLicensed.
- UpdateVersion. Identifies the internal version number of the file.

- UUID. Uniquely identifies the file. Each file has its own UUID.
- Created. Identifies the date and time of file creation.
- CountryISO3. Identifies the country for which the file contains data.
- Type. Contains additional information about the data file type.
- Description. Contains additional information about the data file type.
- DataSetID. Denotes the data set ID value.
- FunctionMinVersion. Denotes the oldest version of Address Verification with which you can use the file.
- PreloadExtent. Identifies the amount of the data that is preloaded into memory. The value can be Full, Partial, or None.
- PreloadSizeBytes. Identifies the amount of data that is preloaded into memory.

FileSetB

Identifies the data files in the FileSetB folder. Each file has a .MD6 extension, and provides the following information:

- FilePath. Identifies the path to the data file on the disk.
- Status. Identifies the status of the file. The status can be Active or NotLicensed.
- UpdateVersion. Identifies the internal version number of the file.
- UUID. Uniquely identifies the file. Each file has its own UUID.
- Created. Identifies the date and time of file creation.
- CountryISO3. Identifies the country for which the file contains data.
- Type. Contains additional information about the data file type.
- Description. Contains additional information about the data file type.
- DataSetID. Denotes the data set ID value.
- FunctionMinVersion. Denotes the oldest version of Address Verification with which you can use the file.
- PreloadExtent. Identifies the amount of the data that is preloaded into memory. The value can be Full, Partial, or None.
- PreloadSizeBytes. Identifies the amount of data that is preloaded into memory.

Formatting the JSON Response

When you configure a function call to the engine, you can include one or more options in the call to determine the format of engine response.

Append the options as a query string segment at the end of the URI. If you do not append an option to the URI, the engine uses the default value for the option.

The following URI includes a range of formatting options:

AV/v1/Jobs/.../Value/IO/Outputs?
HideDefault=true&Format=Smart&MaxLineLength=120&Indent=Spaces&IndentWidth=4&OpenBraceOnSameLine=true&EscapeSolidus=false&CompactEscape=true&Newline=CrLf&BOM=false

The URI options that you can add closely match the options in the InputJSONFormat group that you can set in the IDVEConfig.json file. The InputJSONFormat options determine the format of the JSON that the engine can write to the error logs.

To read a description of the formatting options, see the InputJSONFormat group table under "Core Elements in IDVEConfig.json" on page 13

Initializing the Intelligent Data Verification Engine (IDVE) Framework

Before you can start processing data, you must initialize IDVE. After you process the data, you can call IDVE_Deinitialize() to deinitialize IDVE. You can use the Address Verification function of the IDVE framework to verify the addresses.

Calling Initialization and Deinitialization Functions

Call the IDVE initialization and deinitialization functions in the following sequence:

1. To initialize IDVE, call IDVE_Initialize().

   Pass the configuration information as a string to IDVE_Initialize(). If you use a UTF-16-encoded-string, call IDVE_InitializeW(). You can also pass the configuration information as a file to IDVE_Initialize().

   During initialization, a number of function servers start as separate processes.

2. To deinitialize IDVE, call IDVE_Deinitialize().

Generating a JSON Schema Document

The IDVE_ GetJSONSchema() function generates a JSON schema document for properties that the URI specifies. The IDVE_GetJSONSchema() function returns the data in a UTF-8-encoded string, whereas the IDVE_GetJSONSchemaW() function returns the data in a UTF-16-encoded string. The functions otherwise operate identically. You can call the functions without initializing the engine.

Generating the Schema Document in the UTF-8 Encoding

The following sample code shows the structure of the IDVE_GetJSONSchema() function:

```
IDVE_EXPORTCALL1 IDVE_StatusCode IDVE_EXPORTCALL2 IDVE_GetJSONSchema(
   const char* const kpksURI,
   char* const kpsValueBuffer,
   const IDVE_U64 ku64ValueBufferSize,
   IDVE_U64* const kpu64SizeWritten,
   char* const kpsExtStatusMsg
);
```

The following table shows the parameter definitions for the IDVE_GetJSONSchema() function:

Parameter	Operation	Comment
const char* const kpksURI	[in]	Pointer to the zero-terminated 7-bit ASCII URI, for example "`AV/v1/Jobs/###`". The value might not be NULL.
char* const kpsValueBuffer	[out]	Pointer to the output buffer that receives the UTF-8-encoded JSON value and the terminating zero. The value might not be NULL.
const IDVE_U64 ku64ValueBufferSize	[in]	The size of the output buffer in code units, including the terminating zero.
IDVE_U64* const kpu64SizeWritten	[out]	Pointer to an unsigned 64-bit integer value that receives the size of the output written in code units, excluding the terminating zero. The value can be NULL.
char* const kpsExtStatusMsg	[out]	Pointer to a buffer of size IDVE_EXT_STATUS_MSG_BUFFER_SIZE for an optional extended status message. The value can be NULL.

Generating the Schema Document in the UTF-16 Encoding

The following sample code shows the structure of the IDVE_GetJSONSchemaW() function:

```
IDVE_EXPORTCALL1 IDVE_StatusCode IDVE_EXPORTCALL2 IDVE_GetJSONSchemaW(
   const char* const kpksURI,
   IDVE_WChar* const kpsValueBuffer,
   const IDVE_U64 ku64ValueBufferSize,
   IDVE_U64* const kpu64SizeWritten,
   char* const kpsExtStatusMsg
);
```

The following table shows the parameter definitions of the IDVE_GetJSONSchemaW() function:

Parameter	Operation	Comment
const char* const kpksURI	[in]	Pointer to the zero-terminated 7-bit ASCII URI, for example "`AV/v1/Jobs/###`". The value might not be NULL.
IDVE_WChar* const kpsValueBuffer	[out]	Pointer to the output buffer that receives the UTF-16-encoded JSON value and the terminating zero. The value might not be NULL.
const IDVE_U64 ku64ValueBufferSize	[in]	The size of the output buffer in code units, including the terminating zero.
IDVE_U64* const kpu64SizeWritten	[out]	Pointer to an unsigned 64-bit integer value that receives the size of the output written in code units, excluding the terminating zero. The value can be NULL.
char* const kpsExtStatusMsg	[out]	Pointer to a buffer of size IDVE_EXT_STATUS_MSG_BUFFER_SIZE for an optional extended status message. The value can be NULL.

Reference Databases and Packages

Informatica reference databases are proprietary-format database files that contain reference data for the countries and territories that Informatica supports. The reference databases are read-only and platform-independent.

Types of Reference Data

Address Verification provides the following types of reference database file:

- Address Code Lookup data
- Batch and Interactive data
- Certified data
- QuickCapture data
- GeocodeToAddress data
- Geocoding data
- CAMEO data
- Supplementary data

Reading the Database File Names

The database file names have the following format:

```
<ISO3>_<Function>_<ProcessType>_<ProcessSubType>_<DataSetID>_<MinVersion>.MD6
```

For example, `DEU_ADV_VRF_BIA_001_6_1_0.MD6`

Interpret the file names in the following way:

- ISO3 denotes the three-character ISO country code. For example, DEU for Germany.
- Function denotes the name of the product. In the file name, ADV represents Address Verification.
- ProcessType denotes the type of process. In the file name, VRF represents Verification, ENR represents Enrichment, and RLK represents Reverse Lookup.
- ProcessSubType denotes the process mode for which the file is intended or the type of data that the file contains.

  The following list identifies the subtypes:

  - ACL for AddressCodeLookup mode
  - BIA for Batch and Interactive modes
  - Cxx for Certified mode
  - QCP for QuickCapture mode
  - GTA for GeocodeToAddress mode
  - GAP for arrival point geocoding
  - GRT for rooftop geocoding
  - GST for street center, locality center, and postal code center geocoding
  - CAM for CAMEO data
  - EN1 for supplementary data
- DataSetID denotes the data set ID value. The supported range is 000 through 999.
- MinVersion denotes the oldest version of Address Verification with which you can use the file.

MinVersion has the following format:

```
<MajorVersion>_<MinorVersion>_<ReleaseVersion>
```

Installing the Reference Databases

The *FileSetsDirectoryPath* option in the `IDVEConfig.json` file identifies the root directory for the reference database files and the `FileSetsInfo.json` file. Download the reference database files to a directory named `FileSetA` or `FileSetB` under the root directory. Install the database files directly to `FileSetA` or `FileSetB`. Do not add the files to a subdirectory in either directory.

By default, Address Verification looks for the data files in the *FileSetA* directory. During hot swapping, Address Verification also uses the data files available in the *FileSetB* directory.

Note: If the directory that the FileSetsDirectoryPath option specifies does not permit updates to the FileSetsInfo.json file, you may decide to copy the file to another directory. For more information, see the option descriptions in "Core Elements in IDVEConfig.json" on page 13.

Retrieving the Software Version Number

To fetch the version number of the software, use the IDVE_GetVersion() or IDVE_GetVersionW() function. Use the IDVE_GetVersion() function to retrieve the version number in the UTF-8 character encoding. Use the IDVE_GetVersionW() function to retrieve the version number in the UTF-16 character encoding. The functions otherwise operate identically.

The Address Verification version number count begins with 6 and the IDVE version number count begins with 1 at launch as IDVE is a new module of the software. You can call the functions at any time.

You can retrieve the following version numbers:

- The version of the IDVE framework. When you provide an empty string URI (""), the function fetches the IDVE framework version number, for example "0.0.2.51140".
- The version of the Address Verification function. When you provide an URI to the function ("AV"), the function fetches the Address Verification version number, for example "6.0.0.51140".

Note: You can retrieve the version number without initializing IDVE.

Retrieving the Software Version Number in the UTF-8 Encoding

The following sample code shows the structure of the IDVE_GetVersion() function:

```
IDVE_EXPORTCALL1 IDVE_StatusCode IDVE_EXPORTCALL2 IDVE_GetVersion(
const char* const kpksURI,
char* const kpsValueBuffer,
const IDVE_U64 ku64ValueBufferSize,
IDVE_U64* const kpu64SizeWritten,
char* const kpsExtStatusMsg
);
```

The following table shows the function definitions of the IDVE_GetVersion() function:

Function	Operation	Comment
const char* const kpksURI	[in]	Pointer to the zero-terminated 7-bit ASCII URI with desired version, for example "" for IDVE or "AV" for AddressVerification. The value might not be NULL.
char* const kpsValueBuffer,	[out]	Pointer to the output buffer that receives the UTF-8 encoded version string and the terminating zero. The value might not be NULL.
const IDVE_U64 ku64ValueBufferSize	[in]	Size of the output buffer in code units, including the terminating zero.
IDVE_U64* const kpu64SizeWritten	[out]	Pointer to an unsigned 64-bit integer value that receives the size of the output written in code units, excluding the terminating zero. The value might be NULL.
char* const kpsExtStatusMsg	[out]	Pointer to a buffer of size IDVE_EXT_STATUS_MSG_BUFFER_SIZE for an optional extended status message. The value might be NULL.

Retrieving the Software Version Number in the UTF-16 Encoding

The following sample code shows the structure of the IDVE_GetVersionW() function:

```
IDVE_EXPORTCALL1 IDVE_StatusCode IDVE_EXPORTCALL2 IDVE_GetVersionW(
const char* const kpksURI,
IDVE_WChar* const kpsValueBuffer,
const IDVE_U64 ku64ValueBufferSize,
IDVE_U64* const kpu64SizeWritten,
char* const kpsExtStatusMsg
);
```

The following table shows the function definitions of the IDVE_GetVersionW() function:

Function	Operation	Comment
const char* const kpksURI	[in]	Pointer to the zero-terminated 7-bit ASCII URI with desired version, for example "" for IDVE or "AV" for AddressVerification. The value might not be NULL.
IDVE_WChar* const kpsValueBuffer	[out]	Pointer to the output buffer that receives the UTF-16 encoded version string and the terminating zero. The value might not be NULL.
const IDVE_U64 ku64ValueBufferSize	[in]	Size of the output buffer in code units, including the terminating zero.
IDVE_U64* const kpu64SizeWritten	[out]	Pointer to an unsigned 64-bit integer value that receives the size of the output written in code units, excluding the terminating zero. The value might be NULL.
char* const kpsExtStatusMsg	[out]	Pointer to a buffer of size IDVE_EXT_STATUS_MSG_BUFFER_SIZE for an optional extended status message. The value might be NULL.