OpenEdge Development:
Working with JSON
Contents

Preface ........................................................................................................ Preface–1

1. Using JSON with OpenEdge ................................................................. 1–1
   About JSON ...................................................................................... 1–2
   Benefits of JSON ............................................................................. 1–2
   Benefits of using JSON in OpenEdge .................................................. 1–2
   JSON basics ..................................................................................... 1–3
   JSON data types .............................................................................. 1–3
   Simple values .................................................................................. 1–4
   Complex values .............................................................................. 1–4
   ABL serialization to and from JSON ................................................... 1–5
   Features .......................................................................................... 1–5
   Use cases ....................................................................................... 1–5
   ABL support ................................................................................... 1–6

2. Serializing Temp-tables and ProDataSets to and from JSON .......... 2–1
   ABL support for JSON ....................................................................... 2–2
   Methods and attributes .................................................................... 2–2
   Data type mapping .......................................................................... 2–4
   Writing JSON from a temp-table, temp-table buffer, or ProDataSet .... 2–6
   Writing JSON from a temp-table ..................................................... 2–8
   Writing JSON from a ProDataSet ................................................... 2–9
   Minimizing the size of JSON data .................................................... 2–10
   Reading JSON into a temp-table, temp-table buffer, or ProDataSet ... 2–13
   Reading JSON into a data object with defined schema ..................... 2–15
   Inferring ABL schemas from JSON strings ..................................... 2–16
   Using JSON with WebSpeed ............................................................ 2–20

Index ............................................................................................................ Index–1
## Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1–1:</td>
<td>JSON primitive data types</td>
<td>1–3</td>
</tr>
<tr>
<td>Table 1–2:</td>
<td>JSON complex data types</td>
<td>1–4</td>
</tr>
<tr>
<td>Table 2–1:</td>
<td>JSON methods for temp-tables and ProDataSets</td>
<td>2–2</td>
</tr>
<tr>
<td>Table 2–2:</td>
<td>JSON related attributes for temp-tables and ProDataSets</td>
<td>2–3</td>
</tr>
<tr>
<td>Table 2–3:</td>
<td>Supported non-standard JSON data types</td>
<td>2–4</td>
</tr>
<tr>
<td>Table 2–4:</td>
<td>ABL to JSON data type mapping</td>
<td>2–4</td>
</tr>
<tr>
<td>Table 2–5:</td>
<td>Default initial values for ABL data types</td>
<td>2–11</td>
</tr>
<tr>
<td>Table 2–6:</td>
<td>READ-JS0N( ) method read modes</td>
<td>2–14</td>
</tr>
<tr>
<td>Table 2–7:</td>
<td>Inferring ABL data types from JSON values</td>
<td>2–17</td>
</tr>
</tbody>
</table>
Preface

This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages
- Third party acknowledgements
Purpose

This book provides a programming guide for OpenEdge® developers who want to use JavaScript Object Notation (JSON) with their ABL (Advanced Business Language) applications.

Audience

This book is intended for ABL programmers working with JSON data in their ABL applications.

Organization

Chapter 1, “Using JSON with OpenEdge”

This chapter provides an introduction to JSON and an overview of ABL support for serializing ABL data objects to and from JSON.

Chapter 2, “Serializing Temp-tables and ProDataSets to and from JSON”

This chapter discusses the ABL features that enable you to serialize ABL data objects to and from JSON. It covers data type mapping between ABL and JSON, writing ABL data objects to JSON data, reading JSON data into ABL data objects, and inferring ABL data object schema from JSON data.

Using this manual

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates see the OpenEdge Product Documentation Overview page on PSDN: http://communities.progress.com/pcom/docs/DOC-16074.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the ABL Virtual Machine (AVM). When the documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.
For example, these sentences refer to the ABL compiler’s allowance for parameter passing and the AVM’s possible response to that parameter passing at run time: “ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error.” The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: “The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets.”

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all **UPPERCASE**, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever `integer` appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever `character` appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever `decimal` appears, this is a reference to the `DECIMAL` data type.
- Wherever `numeric` appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This manual uses the following typographical conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bold</td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td><strong>SMALL, BOLD CAPITAL LETTERS</strong></td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td><strong>KEY1+KEY2</strong></td>
<td>A plus sign between key names indicates a <strong>simultaneous</strong> key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td><strong>KEY1 KEY2</strong></td>
<td>A space between key names indicates a <strong>sequential</strong> key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>
### Examples of syntax descriptions

In this example, **ACCUM** is a keyword, and **aggregate** and **expression** are variables:

**Syntax**

```plaintext
ACCUM aggregate expression
```

*FOR* is one of the statements that can end with either a period or a colon, as in this example:

```plaintext
FOR EACH Customer NO-LOCK:
    DISPLAY Customer.Name.
END.
```
In this example, STREAM stream, UNLESS-HIDDEN, and NO-ERROR are optional:

**Syntax**

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

**Syntax**

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

**Syntax**

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one expression or SKIP [ (n) ], and any number of additional expression or SKIP [ ( n ) ] is allowed:

**Syntax**

```
MESSAGE { expression | SKIP [ ( n ) ] } ... 
```

In this example, you must specify {include-file, then optionally any number of argument or &argument-name = "argument-value", and then terminate with }:

**Syntax**

```
{ include-File
  [ argument | &argument-name = "argument-value" ] ... }
```
Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, `WITH` is followed by six optional items:

**Syntax**

```
WITH [ ACCUM max-length ] [ expression DOWN ]
[ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
[ STREAM-IO ]
```

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, `ASSIGN` requires either one or more `field` entries or one `record`. Options available with `field` or `record` are grouped with braces and brackets:

**Syntax**

```
ASSIGN { [ FRAME frame ] { field [ = expression ] } }
[ WHEN expression ] } ...
| { record [ EXCEPT field ... ] }
```

Example procedures

This manual provides numerous example procedures that illustrate syntax and concepts. You can access the example files and details for installing the examples from the following locations:

- The Documentation and Samples located in the `doc_samples` directory on the OpenEdge Product DVD.
- The OpenEdge Product Documentation Overview page on PSDN:

  `http://communities.progress.com/pcom/docs/DOC-16074`

The examples used in Chapter 2, “Serializing Temp-tables and ProDataSets to and from JSON” are found in the `doc_samples\src\prodoc\json` subdirectory.
OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.

- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.

- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.

- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help → Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.

- Choose **Help → Messages** and then type the message number to display a description of a specific OpenEdge message.

- In the Procedure Editor, press the **HELP** key or **F1**.
On UNIX platforms, use the OpenEdge pro command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   \texttt{OpenEdge-install-dir/bin/pro}

2. Press \texttt{F3} to access the menu bar, then choose Help $\rightarrow$ Messages.

3. Type the message number and press \texttt{ENTER}. Details about that message number appear.

4. Press \texttt{F4} to close the message, press \texttt{F3} to access the Procedure Editor menu, and choose File $\rightarrow$ Exit.

### Third party acknowledgements

OpenEdge includes AdventNet - Agent Toolkit licensed from AdventNet, Inc. [http://www.adventnet.com](http://www.adventnet.com). All rights to such copyright material rest with AdventNet.

OpenEdge includes ANTLR (Another Tool for Language Recognition) software Copyright © 2003-2006, Terence Parr. All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes software developed by the Apache Software Foundation ([http://www.apache.org/](http://www.apache.org/)). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML)); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Concurrent Java software Copyright 1994-2000 Sun Microsystems, Inc. All Rights Reserved. Neither the name of or trademarks of Sun may be used to endorse or promote products including or derived from the Java Software technology without specific prior written permission; and Redistributions of source or binary code must contain the above copyright notice, this notice and the following disclaimers: This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN MICROSYSTEMS, INC. AND
ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE
SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN MICROSYSTEMS, INC.
OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR
FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE
DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF
LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE,
EVEN IF SUN MICROSYSTEMS, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

OpenEdge includes DataDirect software Copyright © 1991-2007 Progress Software
Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for
JDBC Type 4 driver); Copyright © 1993-2009 Progress Software Corporation and/or its
subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC); Copyright ©
1988-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights
Reserved. (DataDirect Connect for ODBC); and Copyright © 1988-2007 Progress Software
Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect64
for ODBC).

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC
software, which include ICU software 1.8 and later - Copyright © 1995-2003 International
Business Machines Corporation and others All rights reserved. Permission is hereby granted,
free of charge, to any person obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to do so, provided that the above
copyright notice(s) and this permission notice appear in all copies of the Software and that both
the above copyright notice(s) and this permission notice appear in supporting documentation.

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC
software, which include software developed by the OpenSSL Project for use in the OpenSSL
Toolkit (http://www.openssl.org/). Copyright © 1998-2006 The OpenSSL Project. All rights
reserved. And Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

OpenEdge includes DataDirect products for the Microsoft SQL Server database which contain
a licensed implementation of the Microsoft TDS Protocol.

OpenEdge includes software authored by David M. Gay. Copyright © 1991, 2000, 2001 by
Lucent Technologies (dtoa.c); Copyright © 1991, 1996 by Lucent Technologies (g_fmt.c); and
Copyright © 1991 by Lucent Technologies (rnd_prod.s). Permission to use, copy, modify, and
distribute this software for any purpose without fee is hereby granted, provided that this entire
notice is included in all copies of any software which is or includes a copy or modification of
this software and in all copies of the supporting documentation for such software. THIS
SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE
MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR
PURPOSE.

OpenEdge includes software authored by David M. Gay. Copyright © 1998-2001 by Lucent
Technologies All Rights Reserved (decstrtod.c; strtodg.c); Copyright © 1998, 2000 by Lucent
Technologies All Rights Reserved (decstrtof.c; strtod.c); Copyright © 1998 by Lucent
Technologies All Rights Reserved (dmisc.c; gdtoa.h; gethex.c; gmisc.c; sum.c); Copyright © 1998, 1999 by Lucent Technologies All Rights Reserved (gdtoa.c; misc.c; smisc.c; ulp.c); Copyright © 1998-2000 by Lucent Technologies All Rights Reserved (gdtoaimp.h); Copyright © 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Full copies of these licenses can be found in the installation directory, in the C:/OpenEdge/licenses folder. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

OpenEdge includes http package software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.


OpenEdge includes Infragistics NetAdvantage for .NET v2009 Vol 2 Copyright © 1996-2009 Infragistics, Inc. All rights reserved.

OpenEdge includes JSTL software Copyright 1994-2006 Sun Microsystems, Inc. All Rights Reserved. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes OpenSSL software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written
permission, please contact openssl-core@openssl.org. Products derived from this software may not
be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written
permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT
WARRANTY OF ANY KIND, either express or implied. See the License for the specific
language governing rights and limitations under the License agreement that accompanies the
product.

OpenEdge includes Quartz Enterprise Job Scheduler software Copyright © 2001-2003 James
House. All rights reserved. Software distributed on an “AS IS” basis, WITHOUT
WARRANTY OF ANY KIND, either express or implied. See the License for the specific
language governing rights and limitations under the License agreement that accompanies the
product. This product uses and includes within its distribution, software developed by the
Apache Software Foundation (http://www.apache.org/).

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM
are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright
©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

OpenEdge includes Sonic software, which includes software developed by Apache Software
Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan”, “FOP,” “The Jakarta
Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to
derendorse or promote products derived from the Product without prior written permission. Any
product derived from the Product may not be called “Apache”, nor may “Apache” appear in
their name, without prior written permission. For written permission, please contact
apache@apache.org.

OpenEdge includes Sonic software, which includes software Copyright © 1999 CERN -
European Organization for Nuclear Research. Permission to use, copy, modify, distribute and
sell this software and its documentation for any purpose is hereby granted without fee, provided
that the above copyright notice appear in all copies and that both that copyright notice and this
permission notice appear in supporting documentation. CERN makes no representations about
the suitability of this software for any purpose. It is provided "as is" without expressed or
implied warranty.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project
(http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names
“Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the
Products without prior written permission. For written permission, please contact
info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.

OpenEdge includes Sonic software, which includes software developed by IBM. Copyright ©
1995-2003 International Business Machines Corporation and others. All rights reserved.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
associated documentation files (the "Software"), to deal in the Software without restriction,
including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or
sell copies of the Software, and to permit persons to whom the Software is furnished to do so,
provided that the above copyright notice(s) and this permission notice appear in all copies of the
Software and that both the above copyright notice(s) and this permission notice appear in
supporting documentation. Software distributed on an "AS IS" basis, WITHOUT
WARRANTY OF ANY KIND, either express or implied. See the License for the specific
language governing rights and limitations under the License agreement that accompanies the
product. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright © 2000-2001 ModelObjects Group. All rights reserved. The name “ModelObjects” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “ModelObjects”, nor may “ModelObjects” appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.

OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes Sonic software, which includes software developed by the University Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

OpenEdge includes the UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes Vermont Views Terminal Handling Package software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.
OpenEdge includes XML Tools, which includes versions 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which are available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a “Contributor”. Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a “Contributor”. A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the “License”); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license can also be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

OpenEdge includes XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the “License”); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.

OpenEdge includes YAJL software Copyright 2007, Lloyd Hilaiel. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. Neither the name of Lloyd Hilaiel nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE AUTHOR “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
This chapter introduces the OpenEdge® features that allow developers to use JSON strings in their ABL (Advanced Business Language) applications, as described in the following sections:

- About JSON
- JSON basics
- ABL serialization to and from JSON
About JSON

JavaScript Object Notation (JSON) is a data interchange format created from a subset of JavaScript. The Internet Engineering Task Force’s RFC 4627 describes JSON as follows:

“JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data.”

JSON developers use JSON as an alternative data interchange format to XML. XML is widely used to exchange data in a heterogeneous environment. However, some developers consider XML as too verbose for exchanges between a web browser and a web server as part of a rich internet application. For more information about XML, see OpenEdge Development: Working with XML.

JSON’s appeal as an alternative to XML comes from smaller HTTP messages and less complicated syntax. The smaller messages and simpler syntax can provide a significant performance advantage over XML. As a result, JSON is popular for rich internet applications, especially AJAX applications. JSON’s popularity in AJAX applications makes it ideal for WebSpeed® developers.

Benefits of JSON

JSON offers the same kind of benefits that XML does for exchanging data in a heterogeneous environment, such as the following:

- JSON is self-describing. The syntax and hierarchical structure of the JSON strings can in some cases be interpreted by applications that do not already know what data to expect.

- JSON is simple text. This fact makes it suitable and safe for transferring across platforms and operating systems that do not readily share more complex document types. As text, JSON can also be readily displayed and edited in simple editors.

- JSON is compact. An average JSON string is about two thirds of the size of the same data in XML.

- JSON is easy to learn, easy to read, and easy to understand.

Benefits of using JSON in OpenEdge

Some of the benefits of using JSON in your OpenEdge development work include:

- OpenEdge includes built-in parsers for reading and writing JSON.

- JSON parser functionality is accessed using the same types of ABL objects, methods, and attributes as the built-in XML parsers.

- ABL JSON-enablement features are simple enough to quickly implement basic use cases.

- ABL JSON-enablement features can provide improved performance with, as well as native communication to, AJAX applications.
JSON basics

This section provides a brief overview of JSON with the following topics:

- JSON data types
- Simple values
- Complex values

For more information about JSON, start with the JSON Web site, [http://json.org](http://json.org). This site includes basic information about JSON and links to other sites describing support for JSON in various languages and applications.

**JSON data types**

Because JSON is designed to be lightweight, it supports only four primitive data types, as shown in Table 1–1.

**Table 1–1: JSON primitive data types**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>A string of Unicode characters enclosed in double quotes. A backslash () serves as the escape character.</td>
<td>&quot;jump rope&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>An unquoted numeric value, which can include an exponent</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>54.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9582e-42</td>
</tr>
<tr>
<td>Boolean</td>
<td>An unquoted lowercase literal string of TRUE or FALSE</td>
<td>true</td>
</tr>
<tr>
<td>Null</td>
<td>An unquoted lowercase literal string of NULL</td>
<td>null</td>
</tr>
</tbody>
</table>

The data type of a value is determined by the format of the value. In addition to these primitive data types, there are some non-standard data types in common usage for certain values. For more information about ABL support of non-standard data types, see the “Data type mapping” section on page 2–4.
JSON also supports two complex data types, as shown in Table 1–2:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>A comma-delimited list of named values, either simple or complex, enclosed in braces</td>
<td>{ &quot;myString&quot;: &quot;jump rope&quot;, &quot;myNum&quot;: 17, &quot;myBool&quot;: false }</td>
</tr>
<tr>
<td>Array</td>
<td>A comma-delimited list of unnamed values, either simple or complex, enclosed in brackets</td>
<td>[ &quot;jump rope&quot;, 17, false ]</td>
</tr>
</tbody>
</table>

**Simple values**

A *simple value* is a name/value pair. The name is always quoted and separated from the value by a colon, as shown:

```
"name" : value
```

The exception to this is the list of values in an array, as shown in Table 1–2. In an array, you access a particular value by a numeric index, rather than by a name.

**Complex values**

JSON enables you to name objects and arrays and combine them into complex values. A *complex value* combines simple values to represent more complicated data. The following example shows an object made up of other objects and arrays:

```
{
    "salesRep": { "name": "Dorothy Gale", "age": 38, "region": "Kansas, USA" },
    "tractorSales": { "2009Quarterly": [ 13, 27, 18, 9 ], "2008Quarterly": [ 11, 17, 32, 5 ], "2007Quarterly": [ 9, 25, 16, 10 ] }
}
```

**Note:** The previous JSON example has been formatted for easier reading.
ABL serialization to and from JSON

ABL temp-tables and ProDataSets have the ability to serialize their data to a JSON string. Similarly, you can read JSON data into a temp-table or ProDataSet. These features parallel the features for XML serialization of temp-tables and ProDataSets.

Features

The JSON features of temp-tables and ProDataSets allow you to take advantage of their rich relational features while providing a standards-based method for sharing data with other application. These JSON features include the following:

- Read JSON data to populate an empty temp-table or ProDataSet
- Read JSON data into temp-tables and ProDataSets that already contain data
- Write JSON data from a temp-table or ProDataSet to JSON strings

Unlike the matching XML features, you cannot perform a perfect round-trip of JSON write/reads. If the ABL virtual machine (AVM) parses an incoming JSON string into a dynamic temp-table or ProDataSet that does not already have a schema, the AVM must infer the schema from the data. This process means that you are unlikely to have data objects with identical schema on both sides of a transaction. Meta data, like key fields, is lost in the serialization to and from JSON.

The JSON features are available as attributes and methods on:

- Temp-table objects
- Temp-table buffer objects (acts on the entire temp-table, not just the current buffer contents)
- ProDataSet objects

Use cases

The JSON read and write features are robust and versatile. The following examples demonstrate common use cases that can be solved with the features:

- Provide interoperability between OpenEdge and another JSON-enabled platform or application.
- Use JSON data as a persistent storage mechanism between ABL sessions.
- Simplify or replace existing ABL code that performs JSON reads and writes on temp-tables and ProDataSets.
ABL support

The serialization features are provided by the following methods on temp-tables, temp-table buffers, and ProDataSets:

- **READ-JSON( ) method** — Reads a JSON string into an ABL object
- **WRITE-JSON( ) method** — Writes the contents of the ABL object to a JSON string

You do not need to be familiar with JSON to use these methods and the associated attributes.
The AVM can serialize the data from ABL temp-tables and ProDataSets to a JSON string. Similarly, you can read or read and load JSON data into a temp-table or ProDataSet. This chapter covers the following topics:

- ABL support for JSON
- Data type mapping
- Writing JSON from a temp-table, temp-table buffer, or ProDataSet
- Reading JSON into a temp-table, temp-table buffer, or ProDataSet
- Using JSON with WebSpeed

Note: The examples used in this chapter are found in the doc_samples\src\prodoc\json directory.
ABL support for JSON

The ABL support for serializing data objects to and from JSON parallels the ABL support for serializing those data objects into XML. The ABL support for JSON includes the following:

- Serializing to and from JSON with a pair of methods
- Affecting JSON serialization through attributes on the data object handles
- Nesting temp-tables
- Hiding foreign fields
- Omitting initial values
- Identifying JSON strings through an attribute on the WEB-CONTEXT system handle

The ABL support for JSON does not include the following:

- Serializing schema to and from JSON
- Serializing before images for ProDataSets
- Serializing fields with data types based on ABL class definitions

Methods and attributes

To use the JSON read and write features described in this chapter, use the handle to a temp-table, temp-table buffer, or ProDataSet to access methods and attributes of the object. Table 2–1 describes the available methods.

Table 2–1: JSON methods for temp-tables and ProDataSets

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-JSON( )</td>
<td>Reads a JSON string into a ProDataSet, a temp-table, or a temp-table buffer object.</td>
</tr>
<tr>
<td>WRITE-JSON( )</td>
<td>Writes a JSON string from a ProDataSet, a temp-table, or a temp-table buffer object. In the case of the temp-table buffer object, all the records of the temp-table associated with the buffer object are written to the JSON string—not just the one in the buffer.</td>
</tr>
</tbody>
</table>
Table 2–2 describes the JSON related attributes.

### Table 2–2: JSON related attributes for temp-tables and ProDataSets

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Applies to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREIGN-KEY-HIDDEN(^2)</td>
<td>LOGICAL</td>
<td>Data-relation</td>
<td>Specifies whether the WRITE-JSON() method should hide foreign key fields in the child records of a nested data-relation in a ProDataSet.</td>
</tr>
<tr>
<td>NESTED</td>
<td>LOGICAL</td>
<td>Data-relation</td>
<td>Specifies whether the AVM embeds child rows within a parent row in the JSON. This affects both the data and schema.</td>
</tr>
<tr>
<td>SERIALIZE-HIDDEN</td>
<td>LOGICAL</td>
<td>Buffer-field</td>
<td>Indicates whether this field is written when the temp-table is serialized, for example into JSON or XML. This attribute interacts with the XML-NODE-TYPE attribute.</td>
</tr>
<tr>
<td>SERIALIZE-NAME</td>
<td>CHARACTER</td>
<td>ProDataSet, temp-table, temp-table buffer, temp-table buffer field</td>
<td>Optionally specifies the name of a ProDataSet, a temp-table, a temp-table buffer, or a temp-table buffer-field object as it should appear when serialized, for example into JSON or XML. This attribute interacts with the XML-NODE-NAME attribute.</td>
</tr>
</tbody>
</table>

Note: The SERIALIZE-HIDDEN and SERIALIZE-NAME attributes provide generalized support for serializing data objects to either JSON or XML. These attributes interact with the XML-specific attributes in ABL. For more information, see the descriptions of these attributes in *OpenEdge Development: ABL Reference.*

1. See the “Minimizing the size of JSON data” section on page 2–10 for more information.
Data type mapping

In the “JSON basics” section on page 1–3, you saw that JSON uses only a few primitive data types. In addition, the JSON community has created a few non-standard data types to cover a few other common data types.

Table 2–3 shows the non-standard JSON data types that ABL supports.

Table 2–3: Supported non-standard JSON data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>JSON representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date-time</td>
<td>A JSON string in the ISO 8601 format, “yyyy-mm-ddThh:mm:ss.sss+hh:mm”</td>
</tr>
<tr>
<td>Binary data</td>
<td>A JSON string consisting of the Base64 encoded equivalent of the binary data</td>
</tr>
</tbody>
</table>

For serializing ABL data objects to JSON strings, only the ABL data types that you can assign to a temp-table field are mapped to JSON data types. The exception to this is the ABL CLASS data type. The ABL serialization features do no support serializing fields with data types based on ABL class definitions.

Table 2–4 shows how the AVM maps ABL data types to JSON data types. These mappings apply when the AVM writes ABL data to JSON and when the AVM reads JSON data into an ABL data object with a defined schema.

Table 2–4: ABL to JSON data type mapping

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>JSON data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>string (Base64 encoded)</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>string</td>
</tr>
<tr>
<td>CLOB</td>
<td>string</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>number</td>
</tr>
<tr>
<td>DATE</td>
<td>string (ISO 8601 formatted string of the form &quot;yyyy-mm-dd&quot;)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>string (ISO 8601 formatted string of the form &quot;yyyy-mm-ddThh:mm:ss&quot;)</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>string (ISO 8601 formatted string of the form &quot;yyyy-mm-ddThh:mm:ss+hh:mm&quot;)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>number</td>
</tr>
<tr>
<td>HANDLE</td>
<td>number</td>
</tr>
<tr>
<td>INT64</td>
<td>number</td>
</tr>
<tr>
<td>INTEGER</td>
<td>number</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean (true</td>
</tr>
</tbody>
</table>
Notes: When parsing values to or from JSON, the AVM equates the ABL Unknown value (?) to a JSON null value.

If a JSON string’s value is too long for the ABL data type, the AVM generates an error message and the READ-JSON( ) method returns FALSE.

If a JSON number’s value is out of the ABL data type’s range, the AVM generates an error message and the READ-JSON( ) method returns FALSE.

If you read JSON data into an ABL data object that has a schema, the AVM assumes that the JSON values are formatted appropriately for the data object’s schema. For example, if the AVM reads JSON data into an ABL temp-table field of the DATETIME data type, the AVM assumes that the value is a string in the ISO 8601 format. If the JSON value cannot be converted to the expected ABL data type, READ-JSON( ) generates and error message and returns FALSE.

If you read a JSON string into a dynamic ABL data object that does not have a schema, the AVM infers the schema from the JSON data. When you read JSON into an ABL data object with an inferred schema, the AVM uses different rules to assign ABL data types to JSON values. With the inferred data types, the AVM makes no attempt to determine if the JSON values represent one of the non-standard data types. For example, all quoted values are mapped to the CHARACTER data type. For more information about inferring schema, see the “Inferring ABL schemas from JSON strings” section on page 2–16.
Writing JSON from a temp-table, temp-table buffer, or ProDataSet

The WRITE-JSON( ) method writes a JSON string containing the data of a temp-table or ProDataSet object. If you use the method on a temp-table buffer, it writes the entire contents of the associated temp-table, not just the contents of the buffer. When writing data from a ProDataSet object, WRITE-JSON( ) writes the current version of data in each row of each table in the ProDataSet object. The AVM ignores any before-image data for a ProDataSet. You cannot write a JSON string from a database buffer.

WRITE-JSON( ) does not write the data object’s schema to JSON, because JSON has no standardized support for schema. The lack of schema information means that the JSON string does not explicitly include indexes and data-relations.

The syntax for WRITE-JSON( ) is shown below. The method returns TRUE or FALSE to indicate whether the operation was successful.

Syntax

```
WRITE-JSON ( mode, { file | stream | stream-handle | memptr | longchar } [, formatted [, encoding [ , omit-initial-values ] ] ] )
```

**mode**

A CHARACTER expression that specifies the target for the JSON string. Valid values are “FILE”, “STREAM”, “STREAM-HANDLE”, “MEMPTR”, and “LONGCHAR”.

**file**

A CHARACTER expression that specifies the name of a file to which the AVM writes the JSON string. You can specify an absolute pathname or a pathname relative to the current working directory. If a file with the specified name already exists, the AVM verifies that the file is writeable and overwrites the file.

**stream**

A CHARACTER expression that specifies the name of a stream. If you specify the empty string (""), the AVM writes the JSON string to the default unnamed output stream. For WebSpeed, write the JSON string to the WebSpeed-defined output stream (WEBSTREAM).

**stream-handle**

A HANDLE variable that specifies a stream object handle.

**memptr**

A MEMPTR variable to contain the JSON string in memory. If you do not specify the encoding parameter, the AVM encodes the text written to the MEMPTR as UTF-8. This method allocates the required amount of memory for the JSON string and sets the size of the variable. When you are finished using the MEMPTR, you must free the associated memory, by setting the MEMTER to zero bytes with the SET-SIZE statement.
A \text{LONGCHAR} variable to contain the JSON string in memory.

The AVM saves the JSON string to the \text{LONGCHAR} variable in the code page that corresponds to the character encoding you specify in the \textit{encoding} option. If you do not specify a character encoding for the JSON string, the AVM saves the \text{LONGCHAR} variable in UTF-8.

If the \text{LONGCHAR} variable’s code page is fixed (that is, set using the \texttt{FIX-CODEPAGE} statement) and the fixed code page is not equivalent to the character encoding you specify in the \textit{encoding} option, the \texttt{WRITE-JSON( )} method generates an error and returns \texttt{FALSE}. The JSON string is not saved to the \text{LONGCHAR}.

An optional \texttt{LOGICAL} expression where \texttt{TRUE} directs the AVM to format the JSON string in a hierarchical manner using extra white space, carriage returns, and line feeds. The default value is \texttt{FALSE}. If you specify the Unknown value (\texttt{?}), the method uses the default value of \texttt{FALSE}.

An optional \texttt{CHARACTER} expression that specifies the name of the character encoding the AVM uses to write the JSON string. The default encoding is "UTF-8".

The encoding name must specify a Unicode transformation format. Valid values are "UTF-8", "UTF-16", "UTF-16BE", "UTF-16LE", "UTF-32", "UTF-32BE", and "UTF-32LE".

\textbf{Note:} If you specify the empty string (""\) or the Unknown value (\texttt{?}), the AVM uses the default encoding of UTF-8.

An optional \texttt{LOGICAL} expression where \texttt{TRUE} directs the AVM to exclude temp-table fields containing their initial values from the JSON string, and \texttt{FALSE} directs the AVM to include all temp-table field data in the JSON. The default value is \texttt{FALSE}. If you specify the Unknown value (\texttt{?}), the method uses the default value of \texttt{FALSE}.

For more information about this option, see the “Minimizing the size of JSON data” section on page 2–10.

\textbf{Note:} The examples used in this chapter are found in the \texttt{doc_samples\src\prodoc\json} directory.
Writing JSON from a temp-table

The following code example defines a static temp-table, fills the temp-table object, and writes the temp-table object to a JSON string:

```plaintext
/* write-json-tt.p */
{pi-json-parameterVarDefs.i} /* parameter variable definitions */
{pi-write-json-tt.i} /* ttCust definition */
DEFINE VARIABLE httCust AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturnValue AS LOGICAL NO-UNDO.

httCust = TEMP-TABLE ttCust:HANDLE.

FOR EACH Customer WHERE CustNum < 4:
    CREATE ttCust.
    BUFFER-COPY Customer TO ttCust.
END.

ASSIGN
cTargetType = "FILE"
cFile = "ttCust.json"
lFormatted = TRUE
cEncoding = ?.

lReturnValue = httCust:WRITE-JSON(cTargetType, cFile, lFormatted, cEncoding).
```

The following is an excerpt of the JSON produced by this procedure:

```json
{"ttCust": [  
    {  
        "CustNum": 1,  
        "Country": "USA",  
        "Name": "Lift Tours",  
        "Address": "276 North Drive",  
        "Address2": "",  
        "City": "Burlington",  
        "State": "MA",  
        "PostalCode": "01730",  
        "Contact": "Gloria Shepley",  
        "Phone": "(617) 450-0086",  
        "SalesRep": "HXM",  
        "CreditLimit": 66700.00,  
        "Balance": 903.64,  
        "Terms": "Net30",  
        "Discount": 35,  
        "Comments": "This customer is on credit hold.",  
        "Fax": "",  
        "EmailAddress": ""  
    },  
    {  
        "CustNum": 2,  
        ""  
    },  
    {  
        "CustNum": 3,  
        ""  
    }  
]}
Writing JSON from a ProDataSet

The following code example defines a static ProDataSet object, attaches its data sources, fills the ProDataSet object, and writes the ProDataSet object to a JSON string:

```plaintext
/* write-json-pds1.p */

{pi-json-parameterVarDefs.i} /* parameter variable definitions */
{pi-write-json-pds1.i} /* dsOrderLog definition - no nesting */

DEFINE VARIABLE hdsOrderLog AS HANDLE NO-UNDO.
DEFINE VARIABLE lRetOK AS LOGICAL NO-UNDO.

hdsOrderLog = DATASET dsOrderLog:HANDLE.
DATA-SOURCE dsCustomer:FILL-WHERE-STRING = "WHERE Customer.CustNum = 2 ".
DATASET dsOrderLog:FILL().

ASSIGN
  cTargetType = "file"
  cFile = "dsOrderLog.json"
  lFormatted = TRUE
  cEncoding = ??.

lRetOK = hdsOrderLog:WRITE-JSON(cTargetType, cFile, lFormatted).
```

The following is an excerpt of the JSON produced by this procedure:

```json
{"dsOrderLog": {
  "ttCustomer": [
    {
      "CustNum": 2,
      "..."
    },
  ],
  "ttOrder": [
    {
      "Ordernum": 94,
      "..."
    },
    ...
    {
      "Ordernum": 6070,
      "..."
    }
  ],
  "ttInvoice": [
    {
      "Invoicenum": 94,
      "..."
    },
    {
      "Invoicenum": 124,
      "..."
    }
  ]}
}
Because the ProDataSet definition did not include the NESTED option for the data-relations, the records from each temp-table are presented after each other. If you do not nest child tables, the JSON does not contain the data relation information. The serialization process also loses any information about key columns.

If you run `write-json-pds2.p` which uses a ProDataSet with nested child tables, the resulting JSON looks like this:

```
{"dsOrderLog": {
  "ttCustomer": [
    {
      "CustNum": 2,
      ... "EmailAddress": ",",
      "ttOrder": [
        {
          "Ordernum": 94,
          ... "Carrier": "Standard Mail",
          "ttInvoice": [
            {
              "Invoicenum": 94,
              ...
            }]
        },
        {
          "Ordernum": 125,
          ... "Carrier": "FlyByNight Courier",
          "ttInvoice": [
            {
              "Invoicenum": 124,
              ...
            }
          ]
        },
        ...
      ]
  }
}}
```

**Minimizing the size of JSON data**

When using JSON to transfer large amounts of data, it is always worthwhile to design your schema to minimize the size of the JSON data. Smaller JSON data reduces memory, network bandwidth, and disk usage. The resource savings can be significant for large JSON objects.

There are two features of ABL JSON support that can reduce the size of JSON data:

- For ProDataSets with nested data relations, you can choose to omit the duplicate entries of foreign key fields that appear in each child record nested within a parent record.
- For any ProDataSet or temp-table, you can choose to omit fields from the JSON whose current value is the same as the initial value of that field.
Omitting foreign key fields in nested child records

When working with a nested data relation in a ProDataSet, the fields that define the relationship appear in both the parent record and the child records. Because the child records are contained within the parent record, the foreign key fields in the child records are redundant. The `WRITE-JSON()` method omits these foreign key fields if you:

- Specify the `FOREIGN-KEY-HIDDEN` option on a `DEFINE DATA-RELATION` statement
- Specify the `foreign-key-hidden` argument on the `ADD-RELATION()` method of a Data-relation object
- Set the `FOREIGN-KEY-HIDDEN` attribute of a Data-relation object to `TRUE`

In all three cases, you must also specify the `NESTED` option for the data-relation.

Before using this option, consider carefully whether the consumers of the JSON data will handle the missing key fields appropriately. In ABL, the `READ-JSON()` method automatically populates foreign keys in nested child records with the value in the outer parent record when the foreign key is omitted from the JSON data. Unless you are sure that a non-ABL consumer of the JSON data will do the same, do not use this option in your nested data-relations.

For example, while a Web browser can read the JSON data and populate a JavaScript object, it will create rows in the nested table without the foreign key field.

Omitting fields with initial values

When a field is defined, it gets an initial value either by using the default initial value for the field’s data type or from the `INITIAL` option in the definition statement or `ADD-NEW-FIELD()` method. Table 2–5 shows the default initial values for ABL data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>&quot;&quot; (empty string)</td>
</tr>
<tr>
<td>CLOB</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DATE</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>HANDLE</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>INT64</td>
<td>0</td>
</tr>
<tr>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>No</td>
</tr>
</tbody>
</table>
Omitting initial value fields from your JSON can be useful if:

- Fields with initial values are not important to the business or application logic of your JSON consumer
- The JSON consumer knows how to recreate missing fields and populate them with initial values

To omit these fields, specify TRUE for the optional `omit-initial-values` argument of the `WRITE-JSON()` method.

When working with large ProDataSets, omitting fields containing their initial values can yield smaller JSON values, more efficient network transfers, and performance gains with the `READ-JSON()` and `WRITE-JSON()` methods.

Although using the `omit-initial-values` option can give your application performance and resource use improvements, you must be sure that the consumers of the generated JSON string will correctly handle the JSON. The ABL `READ-JSON()` method always populates created records with initial values from the temp-table or ProDataSet definition. Other applications might not do this.

### Table 2–5: Default initial values for ABL data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Zero-length sequence of bytes</td>
</tr>
<tr>
<td>RECID</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>ROWID</td>
<td>Unknown value (?)</td>
</tr>
</tbody>
</table>
Reading JSON into a temp-table, temp-table buffer, or ProDataSet

The READ-JSON( ) method loads data into static or dynamic temp-table, temp-table buffer, and ProDataSet objects from a JSON string. You cannot read a JSON string into a database buffer. If the ABL data object has a defined schema, the AVM assumes that the JSON values match up with the ABL fields’ data types as shown in Table 2–4. READ-JSON( ) generates an error message and returns FALSE if the JSON value cannot be converted to the expected ABL data type.

For a dynamic ProDataSet or temp-table that is in the CLEAR state, the AVM infers the object’s schema from the data in the JSON value. If a dynamic temp-table is not in the PREPARED or CLEAR state, the method generates an error and returns FALSE. For more information about inferring schema from a JSON string, see the “Inferring ABL schemas from JSON strings” section on page 2–16.

Here is the syntax for READ-JSON( ). The method returns TRUE or FALSE to indicate if the operation was successful.

Syntax

```
READ-JSON ( source-type, { file | memptr | handle | longchar } [, read-mode] )
```

**source-type**

A CHARACTER expression that specifies the source JSON string type. Valid values are “FILE”, “MEMPTR”, “HANDLE”, and “LONGCHAR”.

**file**

A CHARACTER expression that specifies the name of a file. You can specify an absolute pathname or one relative to the current working directory. The AVM verifies that the file exists and is accessible.

**memptr**

A MEMPTR variable that contains the JSON string in memory. The size of the MEMPTR variable must match the size of the JSON string.

**handle**

A HANDLE variable that specifies the WEB-CONTEXT system handle.

This method reads a JSON string from the WebSpeed Transaction Server. The method verifies that the JSON string was posted to the WebSpeed Transaction Server by checking that the handle’s TS-JS-JSON attribute is YES. The method also verifies that ABL is running in a WebSpeed environment.

**longchar**

A LONGCHAR variable that contains the JSON string in memory.
read-mode

A CHARACTER expression that specifies the mode in which this method reads data from the JSON string into a temp-table or a ProDataSet member buffer. The expression must evaluate to “APPEND”, “EMPTY”, “MERGE”, or “REPLACE”. The default value is "MERGE".

Table 2–6 lists the READ-JSON( ) method modes for reading data.

Table 2–6:   READ-JSON( ) method read modes

<table>
<thead>
<tr>
<th>When the mode is . . .</th>
<th>The READ-JSON( ) method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND</td>
<td>Reads data from the JSON string into the ProDataSet or temp-table object by adding new records to the existing records, without performing any record comparisons. If a record from the JSON string exists in the object (that is, it results in a duplicate unique key conflict), the method generates an error message and returns FALSE.</td>
</tr>
<tr>
<td>EMPTY</td>
<td>Empties the contents of the ProDataSet or temp-table object before reading in data from the JSON string.</td>
</tr>
<tr>
<td>MERGE</td>
<td>Reads data from the JSON string into the ProDataSet or temp-table object by merging new records with existing records in the table. If a record from the JSON string exists in the object (that is, it results in a duplicate unique key conflict), the method does not replace the existing record. If the record from the JSON string does not exist in the object, the method creates a new record.</td>
</tr>
<tr>
<td>REPLACE</td>
<td>Reads data from the JSON string into the ProDataSet or temp-table object by merging new records with existing records in the table. If the record from the JSON string exists in the object (that is, it results in a duplicate unique key conflict), the method replaces the existing record with the new record. If the record from the JSON string does not exist in the object, the method creates a new record.</td>
</tr>
</tbody>
</table>

For a static ProDataSet or temp-table, the serialize name or object name must match the corresponding name found in the JSON string. If the names do not match, the AVM generates an error message and the method returns FALSE. The AVM ignores any columns in the JSON string that do not map to temp-table columns. If you use the SERIALIZE-NAME option in the DEFINE DATASET or DEFINE TEMP-TABLE statement, the AVM uses that name for matching, rather than the ABL object name.

Note: While reading JSON data into an ABL data object, the AVM does not respond to ProDataSet events. The AVM also does not track changes to the data in the ProDataSet or temp-table object, meaning that before-image tables are not updated.
Reading JSON into a data object with defined schema

The READ-JSON() method functions in one of two ways, depending on whether or not its target ABL data object has a defined schema. The most predictable results come when you read JSON data into a data object with a defined schema. In this case, the AVM uses the names to match the ABL ProDataSet, temp-tables, and fields to the JSON objects and name/value pairs. It ignores JSON data for any unmatched data object or field. The AVM raises an error if the JSON value does not match the ABL field's data type.

Note: The examples used in this chapter are found in the doc_samples\src\prodoc\json directory.

The following procedure reads the JSON data that was output by write-json-pds2.p into a ProDataSet that uses a subset of the fields in the original temp-table. It then writes the new ProDataSet out as JSON to another file:

```plaintext
/* read-json-pds2.p */

#define TEMP-TABLE ttCustomer NO-UNDO
  FIELD CustNum LIKE Customer.CustNum
  FIELD Name LIKE Customer.Name
  FIELD State LIKE Customer.State.
#define TEMP-TABLE ttOrder NO-UNDO
  FIELD OrderNum LIKE Order.Ordernum
  FIELD CustNum LIKE Order.CustNum
  FIELD ShipDate LIKE Order.ShipDate.
#define TEMP-TABLE ttInvoice NO-UNDO
  FIELD Invoicenum LIKE Invoice.Invoicenum
  FIELD OrderNum LIKE Invoice.OrderNum
  FIELD InvoiceDate LIKE Invoice.InvoiceDate.

#define DATASET dsOrderLog FOR ttCustomer, ttOrder, ttInvoice
  DATA-RELATION CustOrd FOR ttCustomer,
    ttOrder  RELATION-FIELDS(CustNum,CustNum) NESTED
  DATA-RELATION OrdInv FOR ttOrder,
    ttInvoice RELATION-FIELDS(OrderNum,OrderNum) NESTED.

#define VARIABLE hdsOrderLog AS HANDLE NO-UNDO.
#define VARIABLE lRetOK AS LOGICAL NO-UNDO.

hdsOrderLog = DATASET dsOrderLog:HANDLE.

ASSIGN
  cSourceType = "file"
  cFile       = "dsOrderLog2.json"
  cReadMode   = "EMPTY".

lRetOK = hdsOrderLog:READ-JSON(cSourceType, cFile, cReadMode).

ASSIGN
  cTargetType = "file"
  cFile = "dsOrderLog3.json"
  lFormatted = TRUE
  cEncoding = ?.

lRetOK = hdsOrderLog:WRITE-JSON(cTargetType, cFile, lFormatted).

When you compare the JSON files, you see that only the data that fit into the new ProDataSet made it from dsOrderLog2.json to dsOrderLog3.json.
```
Inferring ABL schemas from JSON strings

The READ-JSON( ) method has to infer a schema when its target ABL data object does not have a schema. Unlike XML, JSON does not have a standard schema language. Because the format of each JSON value indicates its data type, the AVM can infer a schema from a JSON string.

Caution: In general, reading JSON into an ABL data object with an inferred schema is less predictable than the alternative. You should consider carefully how this technique might affect the data, especially if you plan to return the data after processing it.

When the AVM has to infer schema for the data object, the AVM makes two passes through the JSON data: one to build the schema and one to fill in the data. On the first pass, the AVM reads all the records before finalizing the schema, which has the following effects:

- When the AVM parses a JSON null, it provisionally assigns a CHARACTER data type to the column. If a subsequent record includes a non-null value for that column, the AVM assigns that data type to the column. In either case, the AVM equates the null value to the Unknown value (?).
- If different rows contain different fields, the final schema includes all the fields.

The AVM infers ABL schema from JSON data using the following guidelines:

- Any JSON object containing an array of objects is a temp-table. The temp-table’s name is the array’s name.
- The entries in an array of objects are the rows of a single temp-table.
- Each name/value pair in a row’s object is a column in the temp-table. The column’s name is the JSON value’s name.
- If a value in a row object is a JSON array, it is an array column. The AVM infers the data type of the array column from the first value in the inner array.
- Any JSON object that is not an array of objects, but that contains at least one object from which the AVM infers a temp-table, is a ProDataSet. The ProDataSet’s name is the JSON object’s name.
- If the AVM encounters an array of objects within another array of objects, the AVM infers it to be a nested temp-table inside the ProDataSet.
- If the AVM infers a temp-table nested within another inferred temp-table, the AVM attempts to create a data-relation between the two tables. If there is only one pair of fields with matching names in the parent and child tables, the AVM uses that field as the data-relation. If there are no matching fields or more than one matching fields between the tables, the AVM raises an error and the READ-JSON( ) method returns FALSE.

Note: If you call READ-JSON( ) on a temp-table object and the AVM infers a nested temp-table, the method generates an error and returns FALSE. If you call READ-JSON( ) on a ProDataSet object and the JSON data contains only a temp-table, the method generates an error and returns FALSE.
Table 2–7 shows how the AVM maps JSON data types to ABL data types when inferring the schema of a temp-table. By comparing Table 2–7 with Table 2–4, you can see that the differences in the data type mapping make it unlikely that an inferred temp-table matches the original object from which the data was read.

**Table 2–7: Inferring ABL data types from JSON values**

<table>
<thead>
<tr>
<th>JSON value</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string¹</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>number²</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>boolean (true</td>
<td>false)</td>
</tr>
<tr>
<td>null</td>
<td>CHARACTER</td>
</tr>
</tbody>
</table>

1. If a JSON string’s value is too long for the ABL data type, the AVM generates an error message and the READ-JSON( ) method returns FALSE.
2. If a JSON number’s value is out of the ABL data type’s range, the AVM generates an error message and the READ-JSON( ) method returns FALSE.

**Note:** When inferring ABL data types, the AVM does not try to determine if a JSON value represents one of the non-standard data types listed in Table 2–3. The AVM infers a JSON value representing a non-standard data type as a JSON string and assigns it a CHARACTER data type in ABL. For example, a JSON string value in ISO 8601 format is interpreted as a CHARACTER field, not a DATETIME field.

As an example for the inferring process, take the following JSON object:

```json
{"ttCust":
  [
  "Name": ["L", "Frank", "Baum"], "CustNum": 1, "GoldStatus": null},
  "Name": ["Alfred", "E", "Newman"], "CustNum": 2, "GoldStatus": false},
  "Name": ["Bullwinkle", "J", "Moose"], "CustNum": 3, "GoldStatus": true}]
```

The JSON object contains an array name “ttCust”. The AVM sees that it is an array of objects and creates a temp-table named “ttCust” to hold the data. The first name/value pair in the row’s object is an array of JSON strings named “Name”. The AVM creates a CHARACTER field, Name, of EXTENT 3 as the temp-table’s first column. The next name/value pair is a JSON number named “CustNum”. The AVM creates a DECIMAL field, CustNum, as the second column. The final name/value pair contains a JSON null named “GoldStatus”. Because the value is a null, the AVM temporarily chooses CHARACTER as the final column’s data type. The AVM then reads the next record and determines that it contains a JSON Boolean for the final pair and creates a LOGICAL field, GoldStatus, as the final column.
The following procedure reads the output from `write-json-pds2.p` into a dynamic ProDataSet, inferring the ProDataSet’s schema from the JSON data. It then outputs the schema and data to another file, so you can examine the results:

```plaintext
#define define_variable hDataset as Handle no-undo.
define_variable lRetOK as logical no-undo.
define_variable hQuery as Handle no-undo.
define_variable hBuffer as Handle no-undo.
define_variable hField as Handle no-undo.
define_variable idx1 as integer no-undo.
define_variable idx2 as integer no-undo.
define_variable idx3 as integer no-undo.
create_dataset hDataset.
output to InferPDS2.out append.
lRetOK = hDataset:READ-JSON("file", "dsOrderLog2.json", "empty").
run displayResults.
delete_object hDataset no-error.
output close.

procedure displayResults:
message "READ-JSON return value: " lRetOK skip.
message skip "** hDataset schema info **" skip.
message "ProDataSet name: " hDataset:name
  "Num-buffers" hDataset:num-buffers.
do idx1 = 1 to hDataset:num-buffers:
hBuffer = hDataset:GET-BUFFER-HANDLE(idx1).
message skip "Buffer " idx1 " Buffer name: " hBuffer:name.
message "Buffer Field info".
do idx2 = 1 to hBuffer:num-fields:
hField = hBuffer:BUFFER-FIELD(idx2).
message "Field name: " hField:name " Data type: " hField:DATA-TYPE
  " Extent: " hField:EXTENT.
end. /* idx2 loop */
end. /* idx1 loop */
message skip "** hDataset data **".
do idx1 = 1 to hDataset:num-buffers:
hBuffer = hDataset:GET-BUFFER-HANDLE(idx1).
message "*** Buffer " hBuffer:name " Data: ***".
create_query hQuery.
hQuery:set-buffers(hBuffer).
hQuery:QUERY-PREPARE("for each " + hBuffer:name).
hQuery:QUERY-OPEN.
hQuery:get-next() no-error.
do while not hQuery:query-off:
  message skip.
do idx2 = 1 to hBuffer:num-fields:
hField = hBuffer:BUFFER-FIELD(idx2).
if hField:extent = 0 then
  message hField:name ": " hField:buffer-value.
else
  message hField:name.
do idx3 = 1 to hField:extent:
  message hField:name ": " hField:buffer-value(idx3).
end. /* idx3 loop */
end. /* idx2 loop */
hQuery:get-next() no-error.
end. /* hQuery loop */
message skip.
delete_object hQuery no-error.
end. /* idx1 loop */
end procedure.
```
The output from write-json-pds2.p is designed with the READ-JS\text{ON}(\ ) method's inferring feature in mind. The ProDataSet's tables have only one possible foreign key between each pair of nested tables. If you ran write-json-pds2.p substituting the following ProDataSet definition for the include file and then ran read-json-infer-pds2.p on the output, the procedure would generate several errors because there are several fields in each table that match fields in the outer tables.

```plaintext
DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE Customer.
DEFINE TEMP-TABLE ttOrder NO-UNDO LIKE Order.
DEFINE TEMP-TABLE ttInvoice NO-UNDO LIKE Invoice.

DEFINE DATASET dsOrderLog FOR ttCustomer, ttOrder, ttInvoice
  DATA-RELATION CustOrd FOR ttCustomer,
    ttOrder  RELATION-FIELDS(CustNum,CustNum) NESTED
  DATA-RELATION OrdInv FOR ttOrder,
    ttInvoice RELATION-FIELDS(OrderNum,OrderNum) NESTED.

DEFINE DATA-SOURCE dsCustomer FOR Customer.
DEFINE DATA-SOURCE dsOrder FOR Order.
DEFINE DATA-SOURCE dsInvoice FOR Invoice.
```
Using JSON with WebSpeed

Because of JSON’s popularity in AJAX applications, WebSpeed applications are an obvious place to use these features. To support using JSON with WebSpeed, the WEB-CONTEXT system handle has the IS-JSON attribute. The IS-JSON attribute indicates whether or not a JSON string was posted to the WebSpeed Transaction Server. The AVM determines this by looking for the following content-type HTTP headers:

- “application/json”
- “text/json”

The exact procedure for communicating with a rich Internet application vary depending on the application. As a general example, you might have something like the following procedure fragment to handle JSON requests in a WebSpeed application:

```plaintext
PROCEDURE process-web-request:

  DEFINE VARIABLE httCust AS HANDLE NO-UNDO.
  DEFINE VARIABLE lRetOK AS LOGICAL NO-UNDO.
  DEFINE VARIABLE lcData  AS LONGCHAR NO-UNDO.

  httCust = TEMP-TABLE ttCust:HANDLE.

  /* 2 */ IF NOT WEB-CONTEXT:IS-JSON THEN
    DO ON ERROR UNDO, LEAVE:
      /* 3 */ RUN populateTT.
      httCust:WRITE-JSON("LONGCHAR", lcData).

  /* Build and return your initial HTML response with the JSON of the initial temp-table values */
  /* 4 */ output-content-type ("text/html":U).
    RUN writeHTML(lcData).
    
  /* 6 */ ELSE DO:
    lRetOK = httCust:READ-JSON("HANDLE", WEB-CONTEXT, "MERGE").
    /* 7 */ IF NOT lRetOK THEN
      DO:
        output-content-type ("text/html":U).
        {&OUT}
          "<html>" U SKIP
          "<head>" U SKIP
          "<title> {&FILE-NAME} </title>" U SKIP
          "</head>" U SKIP
          "<body>" U SKIP
          "<p>Failed to load JSON into data object:</p>" U SKIP
          "<p> + ERROR-STATUS:GET-MESSAGE(1) + "</p>" U SKIP
          "</BODY>" U SKIP
          "</HTML>" U SKIP.
      END.
      /* 8 */ ELSE DO:
        output-content-type ("application/json":U).
        httCust:WRITE-JSON("STREAM", "WebStream").
      END.
  END.
END PROCEDURE.
```
Using the `process-web-request` procedure, a complete Web request might run as follows:

1. You log in, creating a request for the initial page that starts the `process-web-request` procedure on the WebSpeed Transaction Server.

2. The procedure first checks whether or not the request is a JSON string.

3. The initial request is not a JSON string, so the procedure populates the temp-table with the initial records and serializes the temp-table using `WRITE-JSON()`.

4. The procedure then builds a HTML page around the JSON and sends it back to the Web browser.

5. You update some of the records and posts the changes back to the server.

6. Since the new Web request is a JSON string, the procedure tries to merge the updates into the temp-table using `READ-JSON()`.

7. If merging the updates into the temp-table fails, the procedure builds and returns a HTML page reporting the error.

8. If merging the updates into the temp-table succeeds, the procedure serializes the updated temp-table using `WRITE-JSON()` and posts the updates back to the Web browser.
Index

A
ABL support
  data type mapping 2–4
  non-standard JSON data types 2–4
AJAX
  JSON popularity 1–2
Array
  JSON data type 1–4

B
Boolean
  JSON data type 1–3

C
Codepages
  encoding for serialization 2–7
Complex data types
  JSON 1–4
Complex values
  JSON structure 1–4

D
Data types
  array 1–4
  Boolean 1–3
  complex 1–4
  inferred 2–17
  mapping 2–4
  non-standard JSON 2–4
  null 1–3
  number 1–3
  object 1–4
  primitive 1–3
  string 1–3

I
IS-JSON attribute 2–20

J
JavaScript Object Notation (JSON)
  See also  JSON 1–2
JSON
  ABL support for 2–2
  as XML alternative 1–2
  complex data types 1–4
  complex values 1–4
  data type mapping 2–4
  definition 1–2
  inferring schema 2–16
  minimizing output 2–11
  non-standard data types 2–4
  primitive data types 1–3
  serialization related attributes 2–3
  simple values 1–4
  unsupported ABL features 2–2
  use in OpenEdge 1–2
  WebSpeed 2–20
Index

N
Null
JSON data type 1–3
Number
JSON data type 1–3

O
Object
JSON data type 1–4

P
Parsers
included in OpenEdge 1–2
Primitive data types
JSON 1–3
ProDataSets
INITIAL option 2–11
NESTED option 2–10
omitting foreign keys 2–11
reading JSON into 2–13, 2–15
serialization example 2–9
serialization related attributes 2–3
serializing to JSON 2–6

R
READ-JSON( ) method 2–2
data objects with schema 2–15
inferring schema 2–16
syntax 2–13

S
Schema
inferring data types 2–17
inferring 2–16
READ-JSON( ) method 2–15
Serialization
attribute interaction 2–3
codepages 2–7
minimizing output 2–11
related attributes 2–3
Simple values
JSON structure 1–4
String
JSON data type 1–3

T
Temp-tables
INITIAL option 2–11
reading JSON into 2–13
serialization example 2–8
serialization related attributes 2–3
serializing to JSON 2–6

W
WebSpeed
using JSON 2–20
WRITE-JSON( ) method 2–2
syntax 2–6

X
XML
serialization attribute interaction 2–3