PUGCHALLENGE AMERICAS

How to Walk your JSON Data into OpenEdge

By Paul Guggenheim

About PGA

- A Progress Evangelist since 1984, and enlightening Progress programmers since 1986
- Designed several comprehensive Progress courses covering all levels of expertise including - The Keys to OpenEdge®
- **OpenEdge and Sitefinity** Partner
- **White Star** Software Strategic Partner
- **Consultingwerk** Partner
- **AppPro** Reseller
- Major consulting clients include Carrier Logistics, Chicago Metal Rolled Products, Eastern Municipal Water District, Foxwoods Casino, Gordon Food Service, Hendrickson Trailer, Interlocal Pension Fund, International Financial Data Services, National Safety Council, and Stanley Engineering.
- Head of the Chicago Area Progress Users Group
- PUG Challenge Steering Committee Member

Agenda

- JSON Overview
  - JSON Data Types
    - Simple
    - Complex
- Reading/Writing JSON To/From Temp-Tables and ProDataSets
  - Read-JSON Method
  - Write-JSON Method
- Use Built-in JSON Classes to convert Data into Customized Temp-Tables
- Walk the JSON Tree Examples
- PGA JSON Analyzer Demonstration

JSON Overview

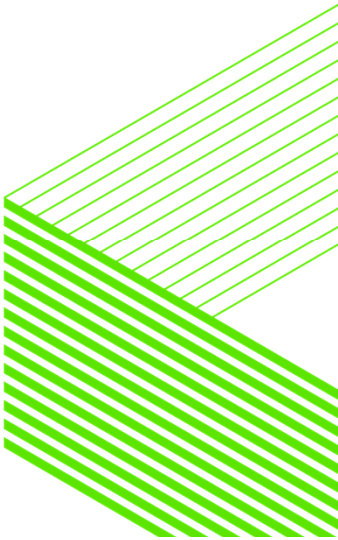
- What is JSON? – JavaScript Object Notation is a lightweight, data interchange format.
- Alternative to XML with a smaller footprint <color>green</color> vs “color”: “green”,
- JSON Features
  - Self Describing
  - Simple Text
  - More Compact resulting in better performance than XML
  - Easy to learn, read and understand
- OpenEdge JSON Support
  - Built-in parsers for reading and writing JSON to Temp-Tables and ProDataSets
  - JSON Classes for performing more sophisticated manipulation

JSON Rules and Data Types

- All JSON files must start with a { and end with a } or a [and a].
- JSON consists of Name Value Pairs – “name”: value
- Simple Data Types
  - String – enclosed in double quotes – “Red Dog”
  - Number – unquoted, may include an exponent – 5.4321e5
  - Boolean – unquoted, lowercase either true or false - true
  - Null – unquoted literal null - null
- Complex Data Types
  - Object – comma-delimited list of name/value pairs, either simple or complex
  - Array – comma-delimited list of unnamed values, either simple or complex

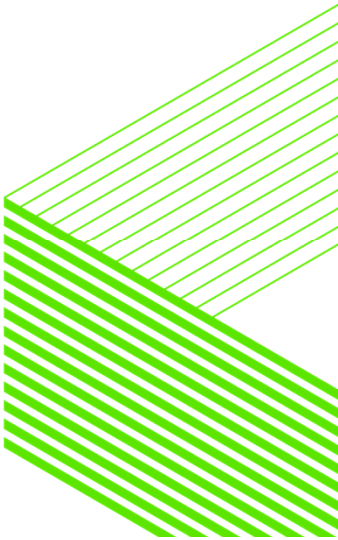
JSON Rules and Data Types – (continued)

- Complex Data Types
  - Object – comma-delimited list of name/value pairs, either simple or complex
    - Example: “car”: { “color”: “black”, “cylinders”: 6, “hybrid”: false }
  - Array – comma-delimited list of unnamed values, either simple or complex
    - Example: “music”: [“rock”, “jazz”, “blues”, “classical”]

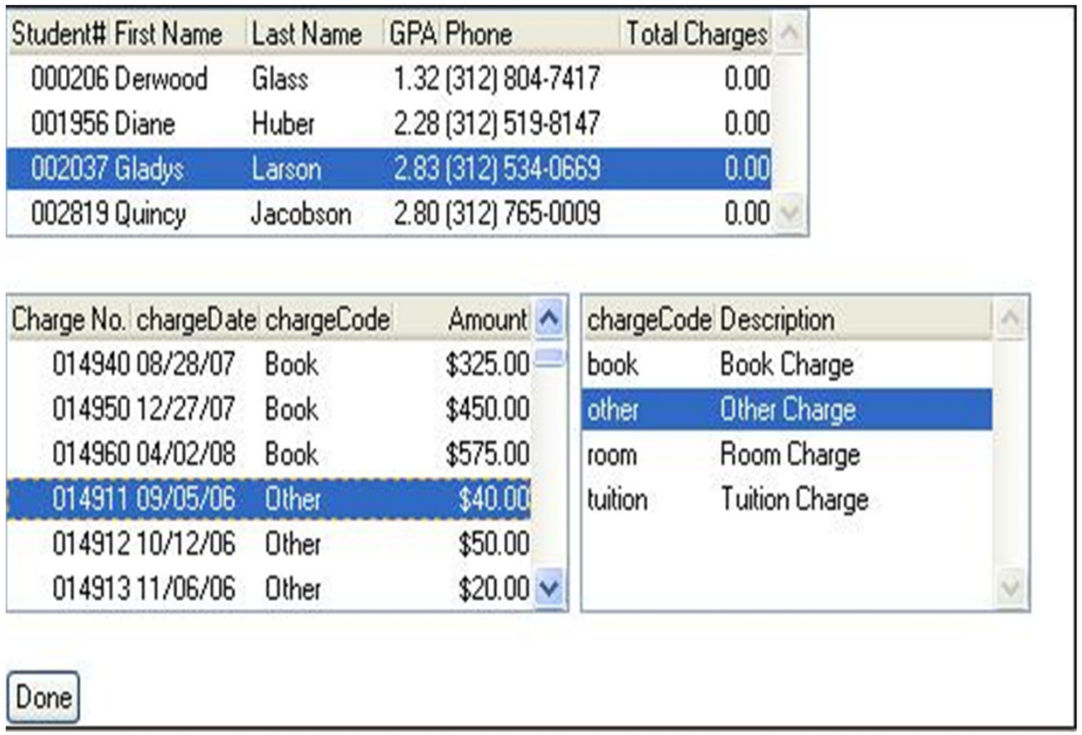


JSON Rules and Data Types – (continued)

- Validate JSON Format
  - The following link will validate the JSON Data: https://jsonlint.com/



Writing JSON from Temp-Tables and ProDataSets



Student#	First Name	Last Name	GPA	Phone	Total Charges
000206	Derwood	Glass	1.32	(312) 804-7417	0.00
001956	Diane	Huber	2.28	(312) 519-8147	0.00
002037	Gladys	Larson	2.83	(312) 534-0669	0.00
002819	Quincy	Jacobson	2.80	(312) 765-0009	0.00

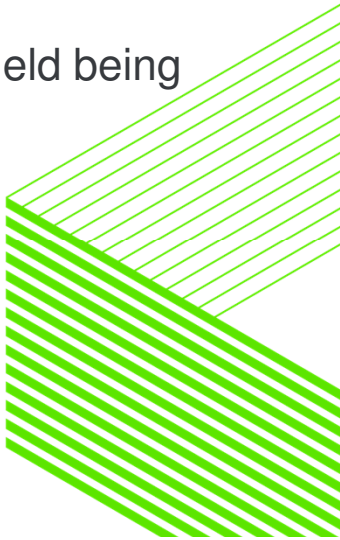
Charge No.	chargeDate	chargeCode	Amount
014940	08/28/07	Book	$325.00
014950	12/27/07	Book	$450.00
014960	04/02/08	Book	$575.00
014911	09/05/06	Other	$40.00
014912	10/12/06	Other	$50.00
014913	11/06/06	Other	$20.00

chargeCode	Description
book	Book Charge
other	Other Charge
room	Room Charge
tuition	Tuition Charge

Done

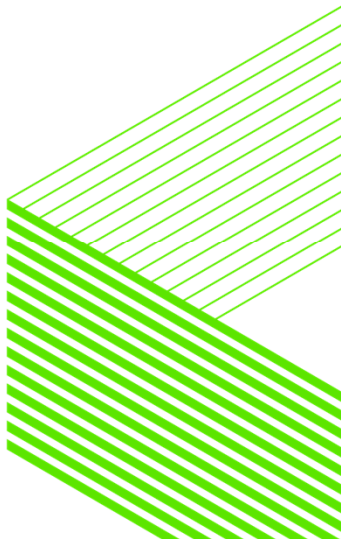
Writing JSON from Temp-Tables and ProDataSets

- Writing JSON from a Multiple Temp-Table ProDataSet
- Three temp-tables are used to represent three database tables:
  - **tstudent** for **student**
  - **tstuchrg** for **stuchrg** (student charge)
  - **tcharge** for **charge** (charge type)
- One **student** record may have many **stuchrg** records, with the **studentid** field being the foreign key in the **stuchrg** table.
- One charge type record may have many **sturchrg** records, with the **chargecode** field being the foreign key in the **stuchrg** table.



Writing JSON from Temp-Tables and ProDataSets

```
define dataset dsstuchrg for tstudent, tstuchrg, tcharge
data-relation stuchrg for tstudent, tstuchrg
relation-fields (studentid, studentid)
data-relation charge for tstuchrg, tcharge
relation-fields (chargecode, chargecode).
.
.
.
buffer tstudent:buffer-field("picture"):SERIALIZE-HIDDEN = true.
dataset dsstuchrg:write-json("file","dsstuchrg.json",true /* formatted */).
buffer tstudent:write-json("file","tstudent.json",true /* formatted */).
buffer tcharge:write-json("file","tcharge.json",true /* formatted */).
find first tstudent.
buffer tstudent:serialize-row("json","file","tstudentrow.json", true /* formatted */).
FIND LAST tstuchrg.
buffer tstuchrg:serialize-row("json","file","tstuchrgrow.json", true /* formatted */).
```



Writing JSON from Temp-Tables and ProDataSets

- SERIALIZED-HIDDEN attribute will prevent BLOBs like the picture field from being dumped.

```
buffer tstudent:buffer-field("picture"):SERIALIZE-HIDDEN = true.
```

- In the statement below, the entire dataset dsstuchrg is written to a file.

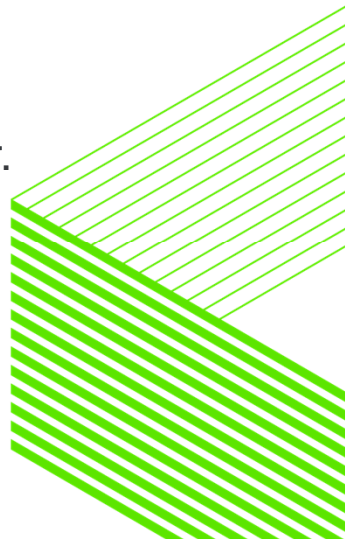
```
dataset dsstuchrg:write-json("file","dsstuchrg.json",true /* formatted */).
```

- An individual temp-table buffer tcharge is written to a file.

```
buffer tcharge:write-json("file","tcharge.json",true /* formatted */).
```

- The SERIALIZE-ROW method exports 1 record from a particular temp-table buffer.

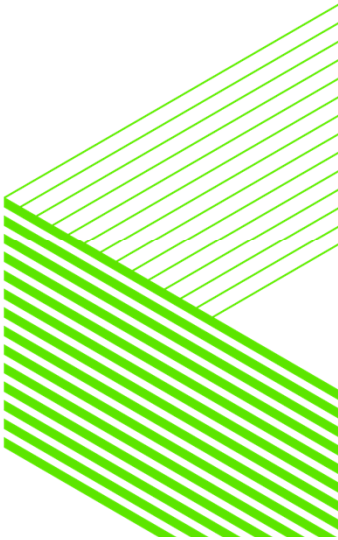
```
find first tstudent.
buffer tstudent:serialize-row("json","file","tstudentrow.json", true /* formatted */).
```



Writing JSON from Temp-Tables and ProDataSets

Tcharge.json:

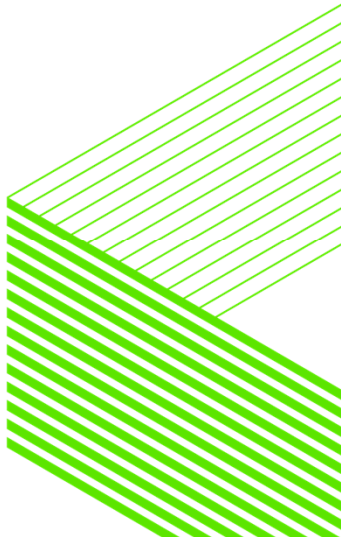
```
{"tcharge": [
  {
    "chargeCode": "book",
    "chargeDescription": "Book Charge"
  },
  {
    "chargeCode": "food",
    "chargeDescription": "Food Charge"
  },
  {
    "chargeCode": "tuition",
    "chargeDescription": "Tuition Charge"
  }
]}
```

Writing JSON from Temp-Tables and ProDataSets

dsstuchrg.json:

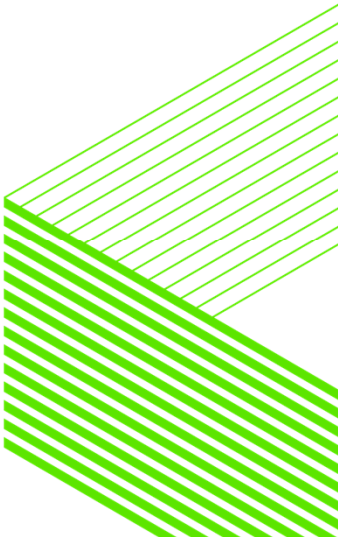
```
{"dsstuchrg": {
  "tstudent": [
    {
      "StudentID": 206,
      "sfirstName": "Derwood",
      "slastName": "Glass",
      "address1": "443 River Avenue",
      "address2": "",
      "address3": "",
      "city": "Chicago",
      "stCode": "IL",
      "postalCode": "60639",
.
.
.
},
```

Writing JSON from Temp-Tables and ProDataSets

tstuchrgrow.json:

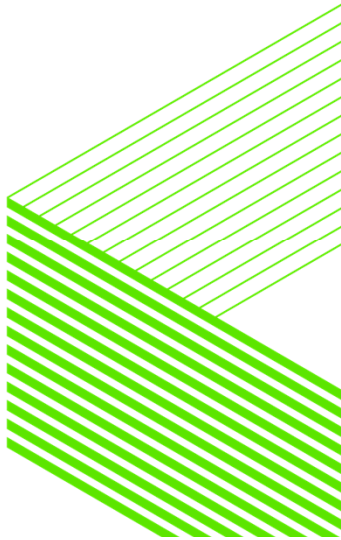
```
{"tstuchrg":   {
    "chargeNo": 117567,
    "studentId": 206,
    "chargeCode": "book",
    "chargeDate": "2017-01-05",
    "chargeAmt": 30.00,
    "studentChargeDescription": ""
  }
}
```



Reading JSON into a static ProDataSet

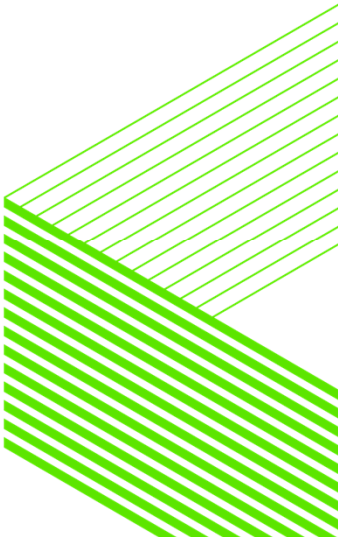
Dsstuchrgreadjson.p:

```
.
.
.
DEFINE DATASET dsstuchrg FOR tstudent, tstuchrg, tcharge
DATA-RELATION stuchrg FOR tstudent, tstuchrg
RELATION-FIELDS (studentid, studentid)
DATA-RELATION charge FOR tstuchrg, tcharge
RELATION-FIELDS (chargecode, chargecode).

DATASET dsstuchrg:READ-JSON ("file", "dsstuchrg.json", "empty").
```



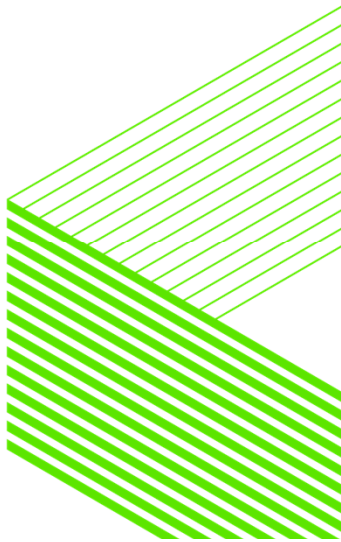
Reading JSON into a static ProDataSet

- To use the READ-JSON method the parameters are:
  1. Source Type such as “file”, “memptr”, “JsonArray” and “JsonObject”
  2. Source Name or variable such as “file name” or variable of type memptr
  3. ProDataSet Read Mode such as “Empty”, “Merge” and “Replace”



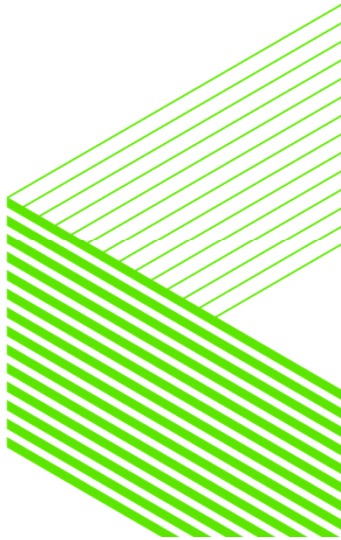
Reading JSON into a dynamic ProDataSet

```
CREATE DATASET DShand.

dshand:READ-JSON("file", "dsstuchrg.json", "empty").

DO i = 1 TO dshand:NUM-BUFFERS WITH FRAME a DOWN STREAM-IO:
    tbuf = dshand:GET-BUFFER-HANDLE(i).

    CREATE QUERY qh.
    qh:SET-BUFFERS(tbuf).

    qh:QUERY-PREPARE("for each " + tbuf:NAME).
    qh:QUERY-OPEN().
    qh:GET-FIRST().
.
.
.
```



Reading JSON into a dynamic ProDataSet

- If it is a *dynamic* ProDataSet, the READ-JSON infers the database schema using a set of rules.
- This is unlike the READ-XML method that reads an explicit XSD file to gather the specific schema definitions. JSON doesn't have a standard schema language.



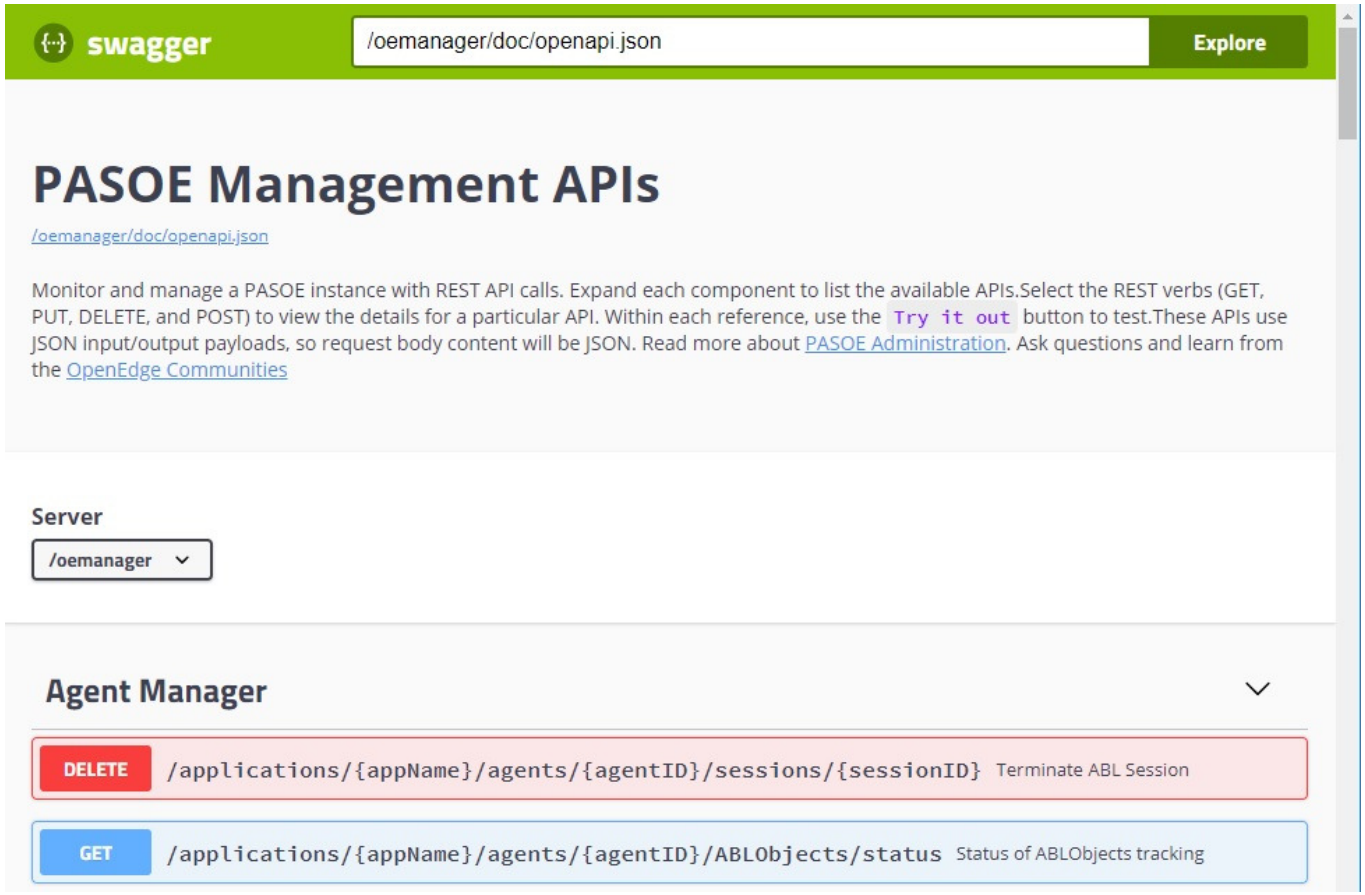
Reading JSON into a dynamic ProDataSet

- Here are some of the guidelines for the AVM inferring ABL Schema:
  - If different rows contain different fields, then the final schema includes all the fields.
  - Any JSON object containing an array of objects is a TEMP-TABLE.
    - The TEMP-TABLE's name is the array's name.
    - Then 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 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.
- Please see page 50-51, in the Working with JSON PDF documentation for more information.

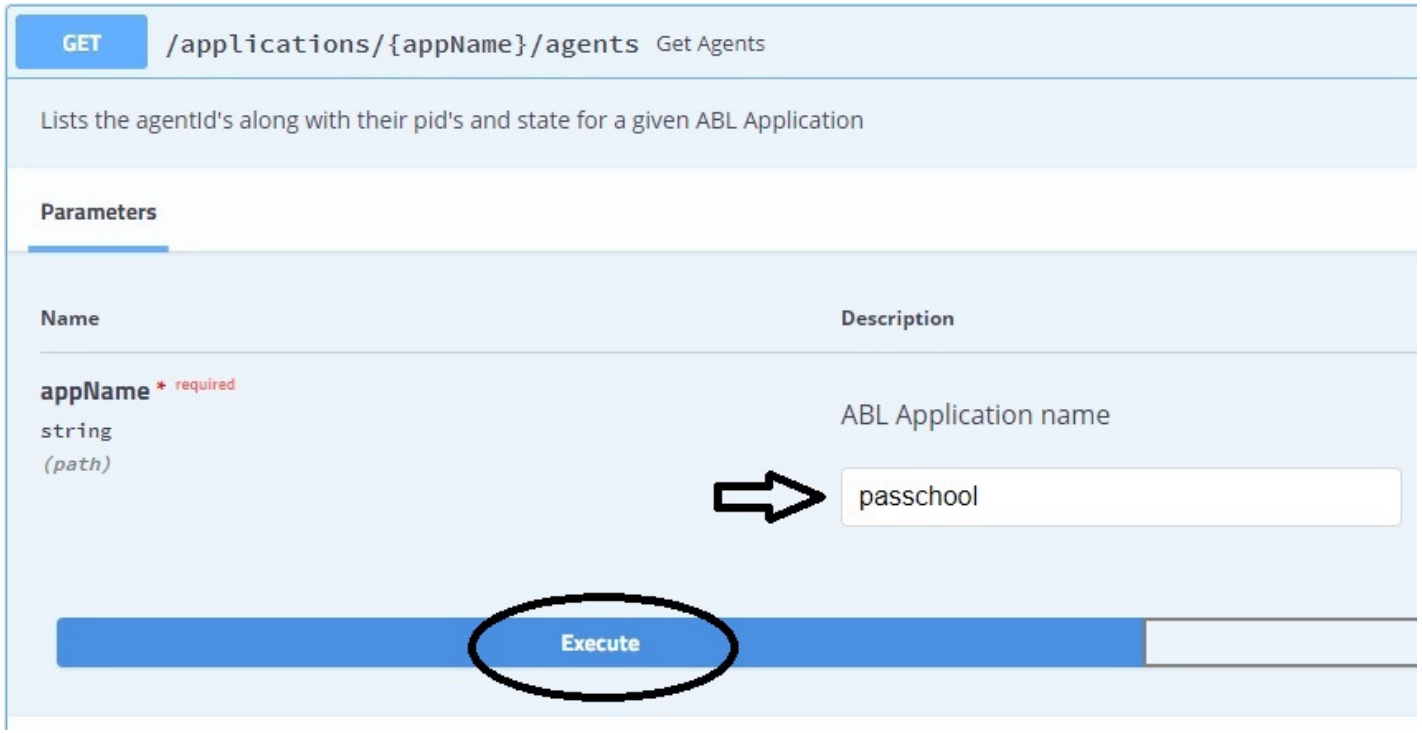
Introducing the new OpenEdge SWAGGER Interface

- SWAGGER is a web API documentation framework.
- In OpenEdge, it is designed to monitor and manage a PASOE instance with REST calls.
- It was released in OpenEdge 11.7.4.
- To access SWAGGER, enter the PASOE instance URL + "/oemanager/".
  - For example, http://localhost:19100/oemanager/
- For documentation on the SWAGGER options type:
  - https://documentation.progress.com/output/ua/OpenEdge_latest/index.html#page/pasoe-admin/rest-api-reference-for-oemanager.war.html

Introducing new SWAGGER Interface

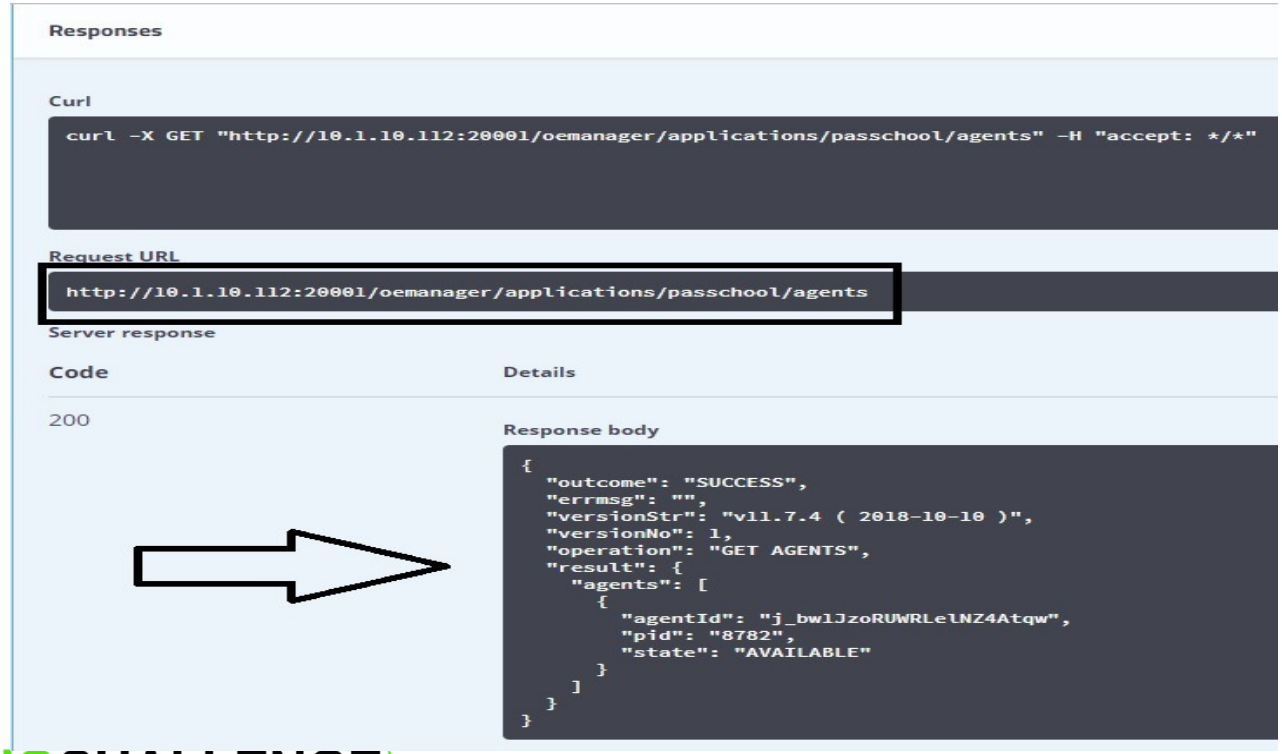


Display Available Agents



Display Available Agents

- The request URL can be entered directly into the browser
- Below is the response, notice the process ID

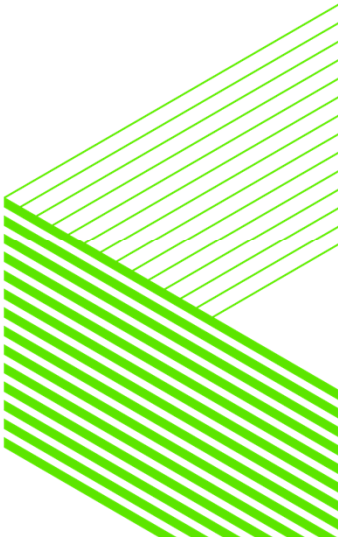


Display Agent Requests

- Use the following URL to get the number of agent requests:

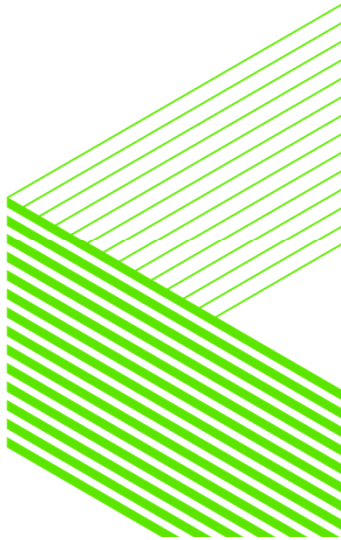
http://10.1.10.112:20001/oemanager/applications/passchool/agents/8782/requests

- Notice the process ID used from the previous query



Display Agent Requests

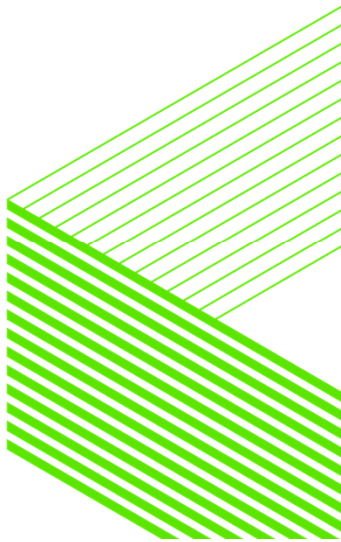

```
{
  "outcome": "SUCCESS",
  "errmsg": "",
  "versionStr": "v11.7.4 ( 2018-10-10 )",
  "versionNo": 1,
  "operation": "",
  "result": {
    "AgentRequest": [
      {
        "RequestProcName": "dspteacher.p",
        "SessionId": 7,
        "ConnectionId": 60,
        "StartTime": "2019-02-10T16:49:46.803",
        "EndTime": "2019-02-10T16:49:46.818",
        "RequestNum": 0
      },
```

Store Request Output in a File, load into Dynamic ProDataSet

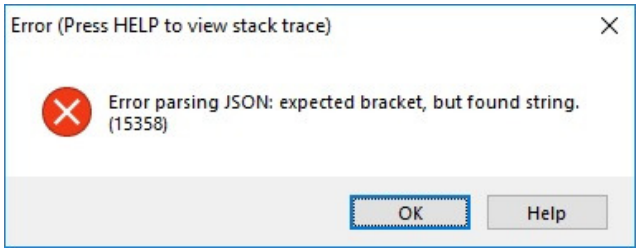
- Store in file agentRequests.json.
- Since the database doesn't know the schema layout for requests, we will use the LOAD-JSON method to load into a dynamic dataset like before.

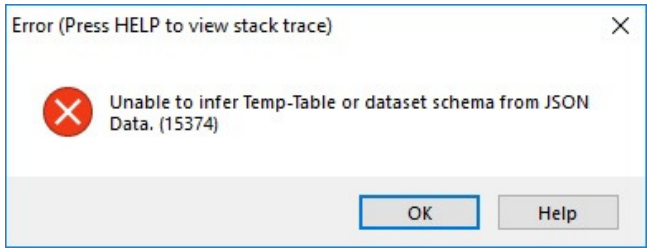
```
CREATE DATASET DShand.
dshand:READ-JSON("file", "agentRequests.json", "empty").
DO i = 1 TO dshand:NUM-BUFFERS WITH FRAME a DOWN STREAM-IO:
    tbuf = dshand:GET-BUFFER-HANDLE(i).
    CREATE QUERY qh.
    qh:SET-BUFFERS(tbuf).
.
.
.
```



Store Request Output in a File, load into Dynamic ProDataSet

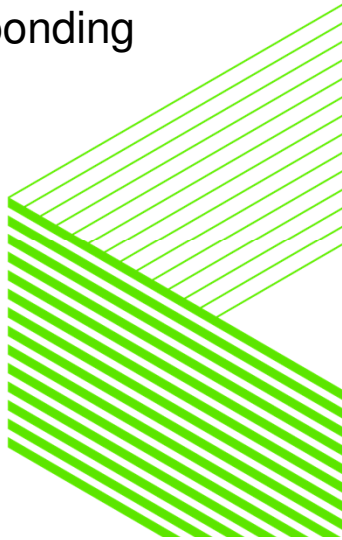
- Store in file agentRequests.json.
- Since the database doesn't know the schema layout for requests, we will use the LOAD-JSON method to load into a dynamic dataset like before.
- Unfortunately, the LOAD-JSON method won't work with this format.
- It is necessary to use Built-in JSON Classes to convert Data into Customized Temp-Tables



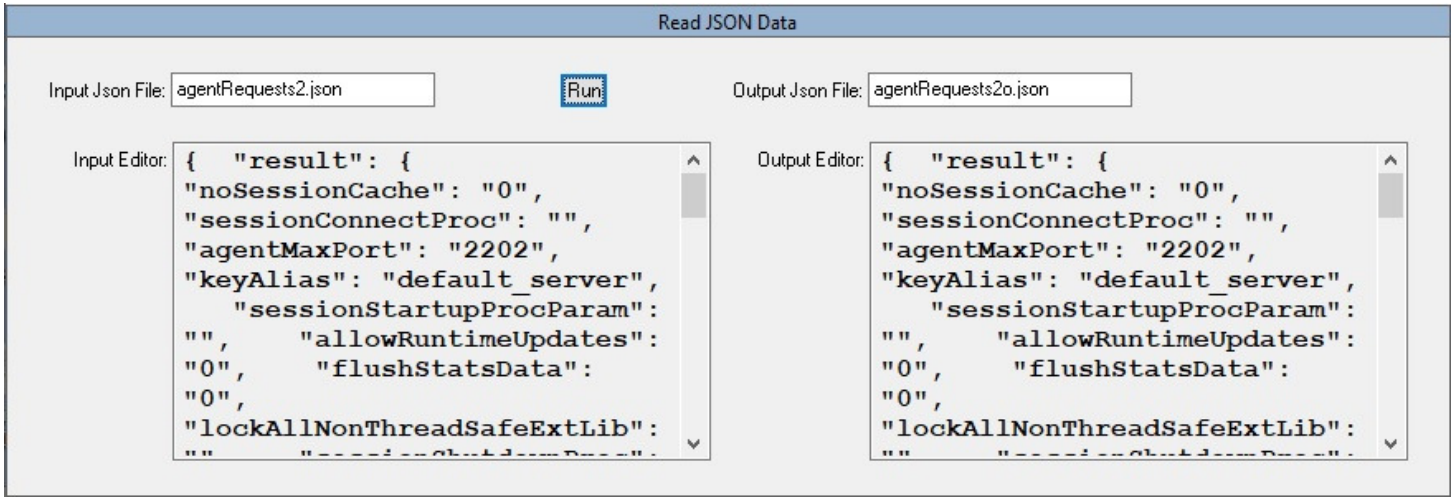


Reading a JSON file into a Built-in JSON Class Object

- Use the ObjectModelParser class to load json data.
- Since we are loading in a file, use the ParseFile method.
- This method returns a JsonConstruct instance.
- The JsonConstruct is an abstract class representing either a JsonObject or JsonArray.
- If the JsonConstruct is a JsonObject, then it is cast into the variable JsonData.
- In readjsondata.p, it is simply written out to json file using the WriteFile method.
- Then the contents of both the input and the output file are loaded into their corresponding editor widgets.
- The editors show that the two files, the input and the output file are identical.



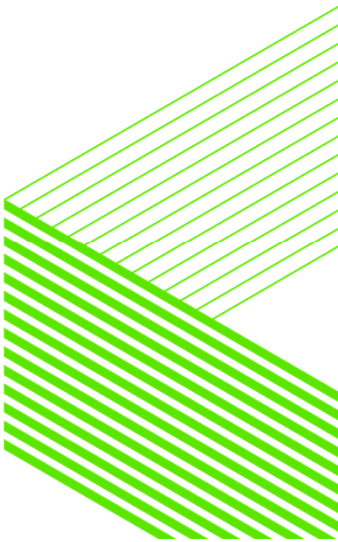
Reading a JSON file into a Built-in JSON Class Object



Reading a JSON file into a Built-in JSON Class Object

Readjsondata.p:

```
oObjectModelParser = NEW ObjectModelParser().
oJsonConstruct = oObjectModelParser:ParseFile(inputfileName).
IF TYPE-OF(oJsonConstruct, "jsonobject") THEN
DO with frame fjson:
  jsondata = CAST(oJsonConstruct,"JsonObject").
  jsonData:WriteFile(outputFileName, TRUE).
  ineditor:read-file(inputfilename).
  outeditor:read-file(outputfilename).
END.
```



Discover JSON Data Types

- After successfully loading Json data into a JsonObject, the next step is to examine the components.
- Use the GetNames() method to do this.
- The GetNames() method returns an array of names. For example,

```
DEFINE VARIABLE jsonData      AS jsonobject.
DEFINE VARIABLE propertyNames AS CHARACTER EXTENT .
DEFINE VARIABLE numprops       AS INTEGER.
ASSIGN propertyNames = jsonData:GetNames()
            numprops = EXTENT(propertyNames).
```

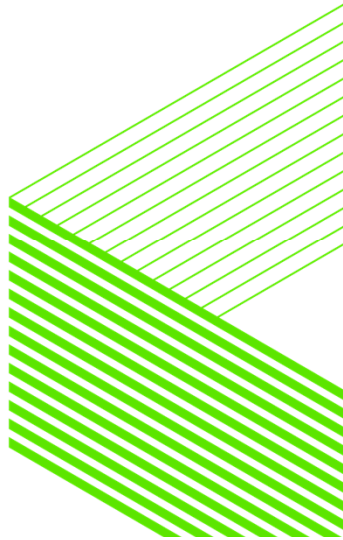
- Since there is no number after EXTENT, PropertyNames is a variable array.
- PropertyNames becomes fixed upon assignment.

Discover JSON Data Types

- The GetType method returns the integer value of Json DataType for a particular component.
- Here is a table showing the Json DataType name to its integer value.

1	2	3	4	5	6
String	Number	Boolean	Object	Array	Null

```
DEFINE VARIABLE dtlist AS CHARACTER NO-UNDO INITIAL
  "String,Number,Boolean,Object,Array,Null".
DO i = 1 TO numprops WITH DOWN:
  DISPLAY propertyNames[i] FORMAT "x(15)"
  ENTRY(jsondata:GetType(propertyNames[i]),dtlist) LABEL "Data Type".
END.
```



Discover JSON Data Types

- These components are outside of the desired request data.
- In jsontraverse1.p, we want the AgentRequest array records, consisting of RequestProcName, SessionId, ConnectionId, StartTime, EndTime and RequestNum.

Input Json File

agentRequests.json

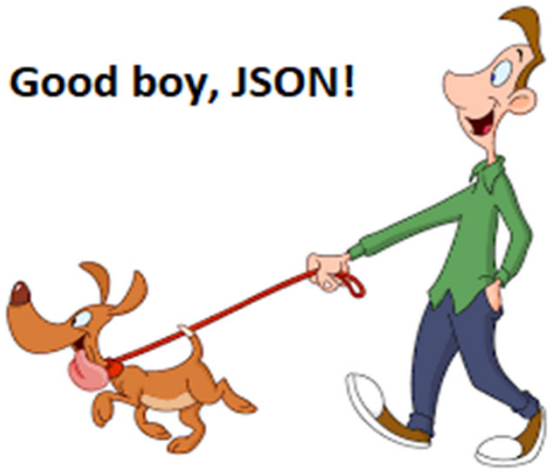
propertyNames	Data Type
outcome	String
errmsg	String
versionStr	String
versionNo	Number
operation	String
result	Object

```
"result": {
    "AgentRequest": [
      {
        "RequestProcName": "dspteacher.p",
        "SessionId": 7,
        "ConnectionId": 60,
        "StartTime": "2019-02-10T16:49:46.803",
        "EndTime": "2019-02-10T16:49:46.818",
        "RequestNum": 0
      },
```

Taking JSON Down the Road

- The GetJsonObject() method will be used to traverse to the object below the current one.
- The GetJsonArray() method will be used to traverse to the array below that one.
- The methods can be piggy-backed together:

```
AgentRequestData = jsondata:GetJsonObject("result"):GetJsonArray("AgentRequest")
```

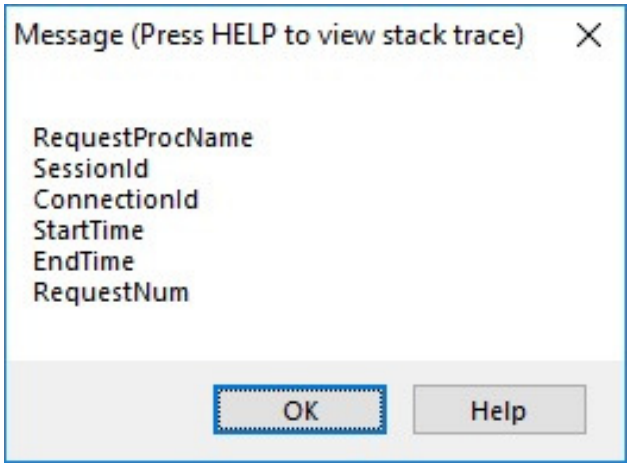


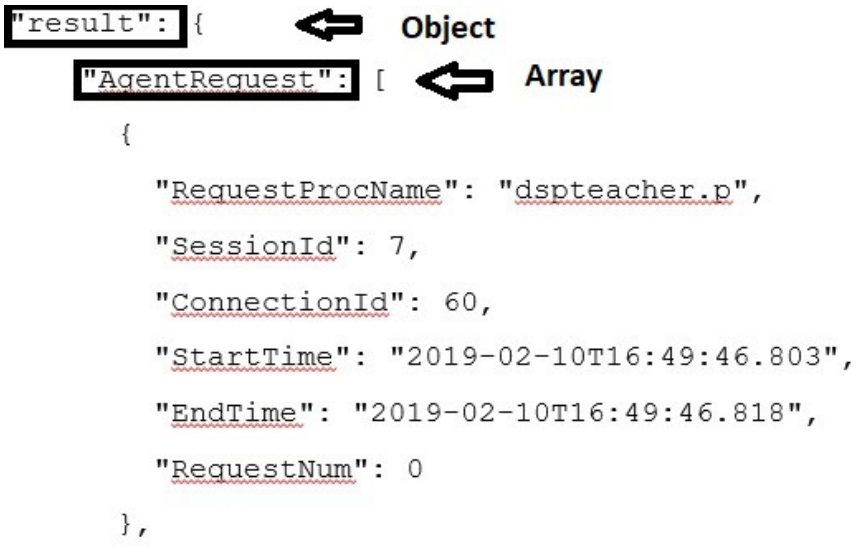

```
"result": {          <= Object
    "AgentRequest": [  <= Array
      {
        "RequestProcName": "dspteacher.p",
        "SessionId": 7,
        "ConnectionId": 60,
        "StartTime": "2019-02-10T16:49:46.803",
        "EndTime": "2019-02-10T16:49:46.818",
        "RequestNum": 0
      },
```

Taking JSON Down the Road (continued)

- The AgentRequest array is an array of objects and is stored in AgentRequestData.
- To get to the object attributes, read the first object of the AgentRequest array.

```
AgentRequestObject = AgentRequestData:GetJsonObject(1)
propertynames2     = AgentRequestObject:GetNames().
```





Taking JSON Down the Road (continued)

- In jsontraverse2.p, we loop through the AgentRequestData array and Populate temp-table tRequest
- Use the getInteger, getCharacter and getDateTimeTZ Methods to retrieve the data

```
DO j = 1 TO AgentRequestData:LENGTH:
  AgentRequestObject = AgentRequestData:GetJsonObject(j).
  CREATE tRequest.
  ASSIGN tRequest.sessionid       = AgentRequestObject:getInteger("SessionId")
         tRequest.RequestNum      = AgentRequestObject:getInteger("RequestNum")
         tRequest.requestprocname = AgentRequestObject:getCharacter("RequestProcName")
         tRequest.starttime       = AgentRequestObject:getDateTimeTZ("StartTime")
         tRequest.endtime         = AgentRequestObject:getDateTimeTZ("EndTime").
END. /* j = 1 TO AgentRequestData:LENGTH */
```

Taking JSON Down the Road (continued)

- Below is the dumped temp-table data

```
7 0 "dspteacher.p" 2019-02-10T16:49:46.803-05:00 2019-02-10T16:49:46.818-05:00
7 1 "dspstuchrg.p" 2019-02-10T16:49:46.836-05:00 2019-02-10T16:49:50.920-05:00
7 2 "dspactivity.p" 2019-02-10T16:49:50.932-05:00 2019-02-10T16:49:50.933-05:00
7 3 "dspcourse.p" 2019-02-10T16:49:50.946-05:00 2019-02-10T16:49:50.952-05:00
7 4 "dspstudent.p" 2019-02-10T16:49:50.962-05:00 2019-02-10T16:49:51.058-05:00
7 5 "dspstuchrg.p" 2019-02-10T16:49:51.065-05:00 2019-02-10T16:49:55.115-05:00
7 6 "dspstuchrg.p" 2019-02-10T16:49:55.123-05:00 2019-02-10T16:49:59.187-05:00
7 7 "dspstudent.p" 2019-02-10T16:49:59.196-05:00 2019-02-10T16:49:59.356-05:00
7 8 "dspemployee.p" 2019-02-10T16:49:59.366-05:00 2019-02-10T16:49:59.366-05:00
7 9 "dspemployee.p" 2019-02-10T16:49:59.374-05:00 2019-02-10T16:49:59.374-05:00
```

What's Next

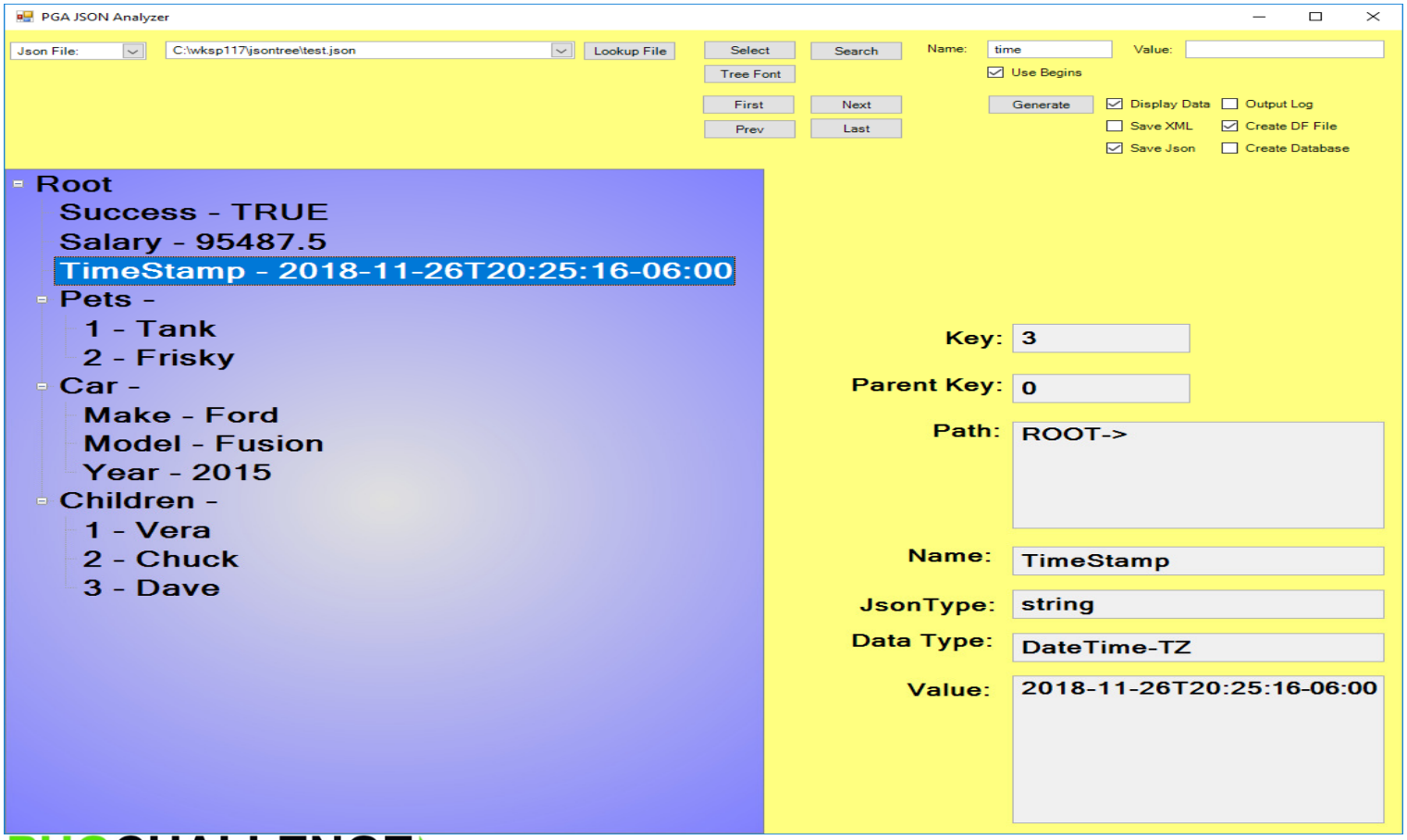
- In the previous example, JSON data was loaded into a static temp-table.
- Using a static temp-table is useful if:
  - The structure of the data is known ahead of time
  - The number of tables and fields being loaded from JSON is small
- What if this is not the case? It is necessary to:
  - Recursively traverse through JSON data to:
  - Dynamically create temp-tables and prodatasets
  - Optionally, generate a corresponding df file
  - Optionally, create a temporary database containing the corresponding database table for the dynamically created temp-tables.

Introducing the new PGA JSON Analyzer

- The PGA JSON Analyzer makes it simpler than ever to connect JSON information to existing OpenEdge applications.
- The tool provides easy upload of JSON files and enables clear viewing options in tree format.
- With a click of button, save data into OpenEdge compatible JSON or XML format files.
- Generate corresponding df files and temporary database related to the dynamically created temp-tables.

PGA JSON Analyzer Demonstration

Introducing the new PGA JSON Analyzer



Summary

- Json Data Types come in both simple and complex forms.
- Reading JSON into Temp-Tables and ProDataSets is easily accomplished using the:
  - Read-JSON Method
  - Provided the JSON was generated using the Write-JSON Method
- If not in the Write-JSON method format, then use Built-in JSON Classes to convert Data into Customized Temp-Tables
- The PGA JSON Analyzer demonstrates how to use OpenEdge's Built-in JSON classes for loading large or complex JSON data into Temp-Tables and ProDataSets.

Questions