PROGRESS OpenEdge

OpenEdge® Development: OpenEdge Mobile 11.5.1 Updates

PROGRESS

Notices

© 2015 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Easyl, Fathom, Icenium, Kendo UI, Making Software Work Together, OpenEdge, Powered by Progress, Progress, Progress Control Tower, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, SpeedScript, Stylus Studio, TeamPulse, Telerik, Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, AppsAlive, AppServer, BravePoint, BusinessEdge, DataDirect Spy, DataDirect SupportLink, , Future Proof, High Performance Integration, Modulus, NativeScript, OpenAccess, Pacific, ProDataSet, Progress Arcade, Progress Pacific, Progress Profiles, Progress Results, Progress RFID, Progress Progress Software, ProVision, PSE Pro, SectorAlliance, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at:
https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx.

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

May 2015

Last updated with new content: Release 11.5.1

Table of Contents

Preface.....7

Purpose.....7
Audience.....8
Using this manual.....8
References to ABL compiler and run-time features.....8
References to ABL data types.....8
Typographical conventions.....9
Examples of syntax descriptions.....10
Long syntax descriptions split across lines.....12
Complex syntax descriptions with both required and optional elements.....12
OpenEdge messages.....12
Obtaining more information about OpenEdge messages.....13

Chapter 1: OpenEdge Mobile 11.5.1 Updates.....15

Creating OpenEdge Mobile Apps using the Telerik Platform.....15
Updating Business Entities for new Mobile Apps in the Telerik Platform.....18
JSDO dialect of the Kendo UI DataSource 25
Session management updates.....31
progress.data.JSDOSession class.....32
JSDOSession methods.....38
addCatalog( ) method (JSDOSession class).....38
login( ) method (JSDOSession class).....41
logout( ) method (JSDOSession class).....45
ping( ) method (JSDOSession class).....47
subscribe( ) method (JSDOSession class).....50
unsubscribe( ) method (JSDOSession class).....51
unsubscribeAll( ) method.....52
JSDOSession properties.....53
authenticationModel property (JSDOSession class).....53
catalogURIs property.....53
clientContextId property.....54
connected property.....54
JSDOs property.....55
loginHttpStatus property.....55
loginResult property.....56
onOpenRequest property.....56
pingInterval property.....58
services property.....59

serviceURI property....60
userName property....61
JSDOSession events....61
offline event....61
online event....63
Note about certificate management for HTTPS testing....65
Behavior changes in the OpenEdge JSDO....65

Preface

For details, see the following topics:

- Purpose
- Audience
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This document describes the support available with the OpenEdge® Service Pack 11.5.1 to create Progress® OpenEdge Mobile Apps in the Telerik® Platform. This includes the requirements for accessing OpenEdge data using the OpenEdge JavaScript Data Object (JSDO) as a remote data service in the Telerik Platform.

As such, it describes the requirements for creating an OpenEdge Business Entity to define a Mobile resource for access by a JSDO in the Telerik Platform and how to access the JSDO as a remote data service using the JSDO dialect of the Kendo UI® DataSource. This dialect of the Kendo UI DataSource binds JSDO data to Kendo UI widgets, such as a Grid or ListView. It also describes how to begin creating the Mobile App within a Telerik AppBuilder Hybrid project based on the Progress Data Services template. This template includes features for creating and using instances of the JSDO dialect of the Kendo UI DataSource.

This document also describes some new session management features to support JSDO access in the Telerik Platform and some new information on managing certificates for testing Mobile Apps using HTTPS. Finally, the document describes a change in behavior since OpenEdge Release 11.5 in how JSDO record change errors are returned for a Mobile resource that supports before-imaging.

Audience

The primary audience for this document is an OpenEdge user or partner who wants to create new OpenEdge Mobile Apps using the Telerik Platform and wants to know how to get started.

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 Progress Communities:

https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx .

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 and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.

Convention	Description
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` end with a period. `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

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

Syntax

```
ACCUM aggregate expression
```

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

```
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 . . . ] }
```

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:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

OpenEdge Mobile 11.5.1 Updates

OpenEdge® Release 11.5.1 includes OpenEdge Mobile documentation updates to describe both OpenEdge Mobile support for developing new Mobile Apps using the Telerik® Platform and a behavior change in how certain record errors are returned to a JSDO for Mobile operations using before-image data.

For details, see the following topics:

- Creating OpenEdge Mobile Apps using the Telerik Platform
- Updating Business Entities for new Mobile Apps in the Telerik Platform
- JSDO dialect of the Kendo UI DataSource
- Session management updates
- Behavior changes in the OpenEdge JSDO

Creating OpenEdge Mobile Apps using the Telerik Platform

The Telerik Platform provides an entirely new environment for developing OpenEdge Mobile Apps with its own project structure, AppBuilder, and UI control set (Kendo UI®). It supports both Web and Hybrid App development using HTML5, JavaScript, jQuery, and Kendo UI, and supports Hybrid Apps using Cordova. OpenEdge Mobile adds support for accessing OpenEdge data from Kendo UI using OpenEdge JavaScript Data Object (JSDO) technology. For more information on the Telerik Platform, see http://docs.telerik.com/platform/help/getting-started/introduction. For more information on Kendo UI, see http://docs.telerik.com/kendo-ui/introduction.

This documentation describes the basic requirements for using Kendo UI features to access OpenEdge data using JSDO technology in new OpenEdge Mobile Apps that you build in the Telerik Platform.

To develop new OpenEdge Mobile Apps using the Telerik Platform, you need to:

- **Manually update OpenEdge Business Entities** — After you create a Business Entity in Progress® Developer Studio for OpenEdge, you need to modify it to be accessed as a Mobile resource from a JSDO that you create in the Telerik Platform. These manual modifications include additional code annotations for the read method, new ABL code for interpreting the read method's `filter` parameter as a JSON Filter Pattern, and adding a new invoke method to provide the total number of records in the result set returned for the single-temp-table dataset parameter or in the parent temp-table of a multi-table dataset parameter.

  For more information, see Updating Business Entities for new Mobile Apps in the Telerik Platform on page 18.

- **Create a Mobile service to generate and deploy a JSDO catalog** — After you create and modify a Business Entity, you need to create a Mobile service for it to generate a JSDO catalog in Developer Studio, and deploy the Mobile service to a Mobile Web application that you can access from the Telerik Platform.

  For more information, see the sections on creating Mobile services in *OpenEdge Development: Mobile Applications* and the online help in Progress Developer Studio for OpenEdge.

- **Create a project for a new Mobile App in the Telerik Platform** — After you have access to a JSDO catalog that defines the Mobile resource you want to access, you can build a new Mobile App to access it in the Telerik Platform. To begin creating this Mobile App in the Telerik Platform:

  1. Create a **Hybrid** app with a blank starting point.
  2. In the new Hybrid app, create an **AppBuilder Hybrid project**.
  3. In the **Create AppBuilder Hybrid project** page, select **Choose project template** and click **Progress Data Service**.
  4. Enter a **Project name** and optional **Description**.
  5. Click **Create project** to create and open the new project in the Telerik AppBuilder.
  6. Open the `README.txt` file in the project to get started.

- **Access OpenEdge data using the JSDO dialect of the Kendo UI DataSource** — As you develop the Mobile App, you will use one or more Kendo UI DataSources to bind a remote data service to Kendo UI widgets, like the Kendo UI Grid. To access OpenEdge data using these Kendo UI widgets, you need to use the *JSDO dialect of the Kendo UI DataSource* described in this document. This dialect of the Kendo UI DataSource is specifically implemented to access a JSDO instance as a remote data service for Kendo UI widgets. However, note that Kendo UI widgets have particular Kendo UI DataSource requirements, and not all of them support the JSDO dialect. The following table summarizes the Kendo UI widgets that support data access through a Kendo UI DataSource and if they support the JSDO dialect.

  Table 1: Kendo UI support for DataSources

Widgets with a DataSource property	Support for the JSDO dialect?	Notes
`ComboBox`	Yes	–

Widgets with a DataSource property	Support for the JSDO dialect?	Notes
`DropDownlist`	Yes	–
`Gantt`	No	Requires `GanttDataSource`
`Grid`	Yes	–
`ListView`	Yes	–
`MultiSelect`	Yes	–
`PivotGrid`	No	Requires `PivotDataSource`
`Scheduler`	No	Requires `SchedulerDataSource`
`TreeList`	No	Requires `TreeListDataSource`
`TreeView`	No	Requires `HierarchicalDataSource`

For an overview and reference to the JSDO dialect of the Kendo UI DataSource, see JSDO dialect of the Kendo UI DataSource on page 25.

- **Manage JSDO login sessions and catalogs for the JSDO dialect of a DataSource using a JSDOSession object** — To create a JSDO dialect of the Kendo UI DataSource, you need to provide access to a JSDO catalog to create the JSDO that the DataSource accesses. As in previous releases, you need to create a user login session to the Mobile Web application that hosts the Mobile service you want to access, and you need to load the JSDO catalog for that Mobile service into your user login session. To support the requirements of the Kendo UI DataSource, it is strongly recommended that you use an instance of the `progress.data.JSDOSession` class to login and load JSDO catalogs. This class provides similar features to the `progress.data.Session` class that you might know from previous releases, but it supports only asynchronous session management methods that return jQuery Promises to handle the results.

  For an overview and reference to this class, see Session management updates on page 31.

Note: The section on session management updates also includes information on managing certificates for HTTPS testing that applies both to Mobile App development on the Telerik Platform as well as to Mobile App development in previous releases of OpenEdge. For more information, see Note about certificate management for HTTPS testing on page 65.

Updating Business Entities for new Mobile Apps in the Telerik Platform

In both current and prior OpenEdge Mobile releases, the Business Entity generated for OpenEdge Mobile assumes that the `filter` parameter of the Mobile read operation can contain a string value of any format that is agreed upon between the developer of the OpenEdge Mobile App and the OpenEdge developer. For an Express project template, Developer Studio generates Business Entity code that looks for an ABL `WHERE` string as the agreed upon value of this parameter, but otherwise leaves it up to the OpenEdge developer to provide the code required to handle a different format for the `filter` value.

To allow the Telerik Kendo UI to access OpenEdge data using the server filtering, sorting, and paging features of the JSDO dialect of the Kendo UI DataSource, you must manually update the ABL code in the Business Entity to handle a new `filter` string format and add a new JSDO invoke method. The Progress Developer Studio for OpenEdge does not currently generate all the annotations to configure the JSDO catalog or the behavior in a Business Entity that are required to provide OpenEdge data to a Kendo UI DataSource using its server features, even when you use an Express Mobile project template. Therefore, you must manually update the Business Entity with additional annotations and code to:

- Handle the `filter` parameter value of the Mobile read operation method as an OpenEdge-proprietary JSON Filter Pattern (described below). This is required to implement the read operation for any JSDO accessed by the Kendo UI DataSource in order to use its server data features.
- Provide a Mobile invoke operation method that returns the total number of records available for access in the OpenEdge database (that is, in the result set) according to the specified server filtering options. This method is required only if the DataSource uses server paging, because Kendo UI needs to know the total number of records available in the server result set for a given DataSource to manage server paging on the client.

The *JSON Filter Pattern* is a simple JSON object that contains the `ablFilter`, `id`, `orderBy`, `skip`, and `top` properties, which are intended to be used as follows:

- **`ablFilter`** — Contains the text of an ABL `WHERE` string (**not** including the `WHERE` keyword itself) on which to filter the OpenEdge database query that returns the result set, as in the following examples:

  - `"ablFilter" : "(State = 'MA') OR (State = 'GA')"`
  - `"ablFilter" : "Name BEGINS 'A'"`

  A value for this property is always specified for server filtering.

  Note: Some Kendo UI widgets make this setting on the Kendo UI DataSource internally.

- **`id`** — Specifies a unique logical ID for data on the server, however you choose to implement it. For example, a string representing the ABL `ROWID` of a record in an OpenEdge database.

  In order for the client to set this `id` property to a value the Business Entity can use to access data, you need to initially return an appropriate value (or multiple values) for each read operation on the data. For example, you might return a value that identifies each row of data in the result set. For OpenEdge data, this can be an `id` field in each temp-table record containing the database `ROWID` of the corresponding record in the OpenEdge database.

Note: The `id` property is not currently used by the Kendo UI, but is used, for example, by Rollbase clients accessing OpenEdge Server Objects.

- **`orderBy`** — A comma-delimited list of the names of fields used to sort the data in the result set (ascending order by default). After any field, `desc` can be added (case-sensitive) to indicate that sort order is descending for that field. For example:
  - `"orderBy" : "Balance, State"` — The data should be sorted ascending, first by `Balance`, then within `Balance`, by `State`.
  - `"orderBy" : "Balance desc"` — The data should be sorted by `Balance` descending.
  - `"orderBy" : "Country, State desc, City"` — The data should be sorted first by `Country` ascending, then within `Country`, by `State` descending, then within `State`, by `City` ascending.

  A value for this property is always specified for server sorting.

  Note: Some Kendo UI widgets make this setting on the Kendo UI DataSource internally.

- **`skip`** — Specifies how many records in the result set to skip before returning (up to) a page of data in the result set. A value for this property is always specified (along with `top`) for server paging. For example, if the requested page size (`top`) is 10 and the request is for the 5th page of data, the value of this property is set to 40.

  Note: The Kendo UI DataSource calculates this value for some Kendo UI widgets internally.

- **`top`** — Specifies how many records should be returned in a single page (that is, the page size) of the result set after using `skip`. A value for this property is always specified (along with `skip`) for server paging. (The final page in the result set can contain a smaller number of records than `top` specifies.)

  Note: Some Kendo UI widgets set this value on the Kendo UI DataSource internally.

The invoke operation method that you need to add in order for server paging to return the total number of records in the server result set must have the following ABL method signature:

Syntax:

```
METHOD PUBLIC VOID countFnName (
  INPUT filter AS CHARACTER,
  OUTPUT numRecs AS INTEGER
)
```

Where:

`countFnName`

Specifies a name for the method. You then assign this name as the value of the `countFnName` configuration property that you specify in the `transport` object when

you instantiate the Kendo UI DataSource in the Mobile App. For more information, see JSDO dialect of the Kendo UI DataSource on page 25.

filter

Is a string containing the same JSON Filter Pattern value that is passed to the read operation method.

numRecs

Is an integer that you assign the total number of records in the server result set identified by *filter*.

Note that the URI annotation for the invoke operation must specify the `filter` input parameter for this method as a URL query parameter, similar to the URI annotation for the read operation, as follows:

Syntax:

```
URI="/countFnName?filter=~{filter~}"
```

Where *countFnName* is the name of your ABL invoke method. Unlike for the read operation, you might have to change the default URI when you use Progress Developer Studio for OpenEdge to annotate this ABL method as an invoke operation. The sample Business Entity (shown later in this section) shows an example of both this method and its annotations.

The Kendo UI DataSource calls this invoke method on the JSDO whenever it calls the `fill( )` method to return a page of data from the server. This method is called **only** when the DataSource has server paging enabled.

Following is a sample Business Entity showing the method generated to implement the Mobile read operation, with manual annotation and code changes required both to implement a JSDO for access by the Kendo UI DataSource and to implement an OpenEdge Server Object for access by a Rollbase client.

This is the include file (`customer.i`) that is referenced by the Business Entity, including a ProDataSet™ (`dsCustomer`) that contains a single temp-table (`ttCustomer`), with fields that you add to the fields that correspond to the existing database table fields indicated in bold, and an additional index you must also add shown in bold:

```
DEFINE TEMP-TABLE ttCustomer BEFORE-TABLE bttCustomer
FIELD id            AS CHARACTER
FIELD seq           AS INTEGER
FIELD CustNum       AS INTEGER      INITIAL "0" LABEL "Cust Num"
FIELD Name          AS CHARACTER    LABEL "Name"
FIELD Address       AS CHARACTER    LABEL "Address"
FIELD Address2      AS CHARACTER    LABEL "Address2"
FIELD Balance       AS DECIMAL      INITIAL "0" LABEL "Balance"
FIELD City          AS CHARACTER    LABEL "City"
FIELD Comments      AS CHARACTER    LABEL "Comments"
FIELD Contact       AS CHARACTER    LABEL "Contact"
FIELD Country       AS CHARACTER    INITIAL "USA" LABEL "Country"
FIELD CreditLimit   AS DECIMAL      INITIAL "1500" LABEL "Credit Limit"
FIELD Discount      AS INTEGER      INITIAL "0" LABEL "Discount"
FIELD EmailAddress  AS CHARACTER    LABEL "Email"
FIELD Fax           AS CHARACTER    LABEL "Fax"
FIELD Phone         AS CHARACTER    LABEL "Phone"
FIELD PostalCode    AS CHARACTER    LABEL "Postal Code"
FIELD SalesRep      AS CHARACTER    LABEL "Sales Rep"
```

```
FIELD State         AS CHARACTER    LABEL "State"
FIELD Terms         AS CHARACTER    INITIAL "Net30" LABEL "Terms"
INDEX seq IS PRIMARY UNIQUE seq
INDEX CustNum IS UNIQUE CustNum
.

DEFINE DATASET dsCustomer FOR ttCustomer.
```

The `id` field is added to each temp-table record to support access to the JSDO as an OpenEdge Server Object by a Rollbase client.

The `seq` field is used to guarantee the order of records in the serialized temp-table that is returned as JSON to the JSDO. You must add the index on `seq` that is both `PRIMARY` and `UNIQUE`. You can also have additional indexes, which can be the same or different than those in the database, as shown for `CustNum`, but the index on `seq` **must** be the `PRIMARY` one.

Following is the class file for the Business Entity, `Customer.cls`.Manually added annotations and code are in bold, **except** in the case of added methods, where only the first and last lines of the method are in bold:

```
@program FILE(name="Customer.cls", module="AppServer").
@openapi.openedge.export FILE(type="REST", executionMode="singleton",
useReturnValue="false", writeDataSetBeforeImage="false").
@progress.service.resource FILE(name="Customer", URI="/Customer",
schemaName="dsCustomer", schemaFile="Customer/AppServer/customer.i").

USING Progress.Lang.*.

USING OpenEdge.BusinessLogic.BusinessEntity.
USING Progress.Json.ObjectModel.*.

BLOCK-LEVEL ON ERROR UNDO, THROW.

CLASS Customer INHERITS BusinessEntity:

    {"customer.i"}

    DEFINE DATA-SOURCE srcCustomer FOR Customer.

    DEFINE VARIABLE iSeq            AS INTEGER      NO-UNDO.

    CONSTRUCTOR PUBLIC Customer():

        DEFINE VAR hDataSourceArray AS HANDLE NO-UNDO EXTENT 1.
        DEFINE VAR cSkipListArray AS CHAR NO-UNDO EXTENT 1.

        SUPER (DATASET dsCustomer:HANDLE).

        /* Data Source for each table in dataset.
           Should be in table order as defined in DataSet */
        hDataSourceArray[1] =  DATA-SOURCE srcCustomer:HANDLE.

        /* Skip-list entry array for each table in DataSet.
           Should be in temp-table order as defined in DataSet */
        /* Each skip-list entry is a comma-separated list of field names
           to be ignored in the CREATE statement */

        cSkipListArray[1] = "CustNum".

        THIS-OBJECT:ProDataSource = hDataSourceArray.
        THIS-OBJECT:SkipList = cSkipListArray.

    END CONSTRUCTOR.
```

```
    @openapi.openedge.export(type="REST", useReturnValue="false",
writeDataSetBeforeImage="true").
    @progress.service.resourceMapping(type="REST", operation="read",
URI="?filter=~{filter~}", alias="", mediaType="application/json").
    @openapi.openedge.method.property (name="mappingType", value="JFP").
    @openapi.openedge.method.property (name="capabilities",
value="ablFilter,top,skip,id,orderBy").
    METHOD PUBLIC VOID ReadCustomer(
            INPUT filter AS CHARACTER,
            OUTPUT DATASET dsCustomer):

      IF filter BEGINS "~{" THEN
        THIS-OBJECT:JFPFillMethod (INPUT filter).
      ELSE DO:
        BUFFER ttCustomer:HANDLE:BATCH-SIZE = 0.
        SUPER:ReadData(filter).
      END.
    END METHOD.

    /* Other CUD and Submit operation methods */
    . . .

    METHOD PRIVATE VOID JFPFillMethod(INPUT filter AS CHARACTER):

      DEFINE VARIABLE jsonParser     AS ObjectModelParser      NO-UNDO.
      DEFINE VARIABLE jsonObject     AS JsonObject             NO-UNDO.
      DEFINE VARIABLE cWhere         AS CHARACTER              NO-UNDO.
      DEFINE VARIABLE hQuery         AS HANDLE                 NO-UNDO.
      DEFINE VARIABLE lUseReposition AS LOGICAL                NO-UNDO.
      DEFINE VARIABLE iCount         AS INTEGER                NO-UNDO.
      DEFINE VARIABLE ablFilter      AS CHARACTER              NO-UNDO.
      DEFINE VARIABLE id             AS CHARACTER INITIAL ?    NO-UNDO.
      DEFINE VARIABLE iMaxRows       AS INTEGER   INITIAL ?    NO-UNDO.
      DEFINE VARIABLE iSkipRows      AS INTEGER   INITIAL ?    NO-UNDO.
      DEFINE VARIABLE cOrderBy       AS CHARACTER INITIAL ""   NO-UNDO.

      /* purge any existing data */
      EMPTY TEMP-TABLE ttCustomer.

      jsonParser  = NEW ObjectModelParser().
      jsonObject  = CAST(jsonParser:Parse(filter), jsonObject).
      iMaxRows    = jsonObject:GetInteger("top")  NO-ERROR.
      iSkipRows   = jsonObject:GetInteger("skip") NO-ERROR.
      ablFilter   = jsonObject:GetCharacter("ablFilter") NO-ERROR.
      id          = jsonObject:GetCharacter("id") NO-ERROR.
      cOrderBy    = jsonObject:GetCharacter("orderBy") NO-ERROR.
      cWhere      = "WHERE " + ablFilter NO-ERROR.

      IF cOrderBy > "" THEN DO:
        cOrderBy = REPLACE(cOrderBy, ",", " by ").
        cOrderBy = "by " + cOrderBy + " ".
        /* NOTE: id and seq fields should be removed from
           cWhere and cOrderBy */
        cOrderBy = REPLACE(cOrderBy, "by id desc", "").
        cOrderBy = REPLACE(cOrderBy, "by id ", "").
        cOrderBy = REPLACE(cOrderBy, "by seq desc", "").
        cOrderBy = REPLACE(cOrderBy, "by seq ", "").
      END.

      lUseReposition = iSkipRows <> ?.

      IF iMaxRows <> ? AND iMaxRows > 0 THEN DO:
        BUFFER ttCustomer:HANDLE:BATCH-SIZE = iMaxRows.
      END.
      ELSE DO:
        IF id > "" THEN
            BUFFER ttCustomer:HANDLE:BATCH-SIZE = 1.
        ELSE
```

```
            BUFFER ttCustomer:HANDLE:BATCH-SIZE = 0.
      END.

      BUFFER ttCustomer:ATTACH-DATA-SOURCE(DATA-SOURCE srcCustomer:HANDLE).

      IF cOrderBy = ? THEN cOrderBy = "".
      cWhere = IF cWhere > "" THEN (cWhere + " " + cOrderBy)
               ELSE ("WHERE " + cOrderBy).
      DATA-SOURCE srcCustomer:FILL-WHERE-STRING = cWhere.

      IF lUseReposition THEN DO:
        hQuery = DATA-SOURCE srcCustomer:QUERY.
        hQuery:QUERY-OPEN.

        IF id > "" AND id <> "?" THEN DO:
          hQuery:REPOSITION-TO-ROWID(TO-ROWID(id)).
        END.
        ELSE IF iSkipRows <> ? AND iSkipRows > 0 THEN DO:
          hQuery:REPOSITION-TO-ROW(iSkipRows).
          IF NOT AVAILABLE Customer THEN
            hQuery:GET-NEXT() NO-ERROR.
        END.

        iCount = 0.
        REPEAT WHILE NOT hQuery:QUERY-OFF-END AND iCount < iMaxRows:
          hQuery:GET-NEXT () NO-ERROR.
          IF AVAILABLE Customer THEN DO:
            CREATE ttCustomer.
            BUFFER-COPY Customer TO ttCustomer.
            ASSIGN  ttCustomer.id  = STRING(ROWID(Customer))
                    iSeq = iSeq + 1
                    ttCustomer.seq = iSeq.
          END.
          iCount = iCount + 1.
        END.
      END.
      ELSE DO:
        IF id > "" THEN DATA-SOURCE srcCustomer:RESTART-ROWID(1)
                        = TO-ROWID ((id)).
        BUFFER ttCustomer:SET-CALLBACK ("AFTER-ROW-FILL", "AddIdField").
        DATASET dsCustomer:FILL().
      END.

      FINALLY:
        BUFFER ttCustomer:DETACH-DATA-SOURCE().
      END FINALLY.

    END METHOD.

    METHOD PUBLIC VOID AddIdField (INPUT DATASET dsCustomer):
      ASSIGN  ttCustomer.id = STRING(ROWID(Customer))
              iSeq = iSeq + 1
              ttCustomer.seq = iSeq.
    END.

    @openapi.openedge.export(type="REST", useReturnValue="false",
writeDataSetBeforeImage="false").
    @progress.service.resourceMapping(type="REST", operation="invoke",
URI="/count?filter=~{filter~}", alias="", mediaType="application/json").
    METHOD PUBLIC VOID count( INPUT filter AS CHARACTER, OUTPUT numRecs AS
INTEGER):
        DEFINE VARIABLE jsonParser   AS ObjectModelParser   NO-UNDO.
        DEFINE VARIABLE jsonObject   AS JsonObject          NO-UNDO.
        DEFINE VARIABLE ablFilter    AS CHARACTER           NO-UNDO.
        DEFINE VARIABLE cWhere       AS CHARACTER           NO-UNDO.
        DEFINE VARIABLE qh           AS HANDLE              NO-UNDO.

        IF filter BEGINS "WHERE " THEN
```

```
            cWhere = filter.
        ELSE IF filter BEGINS "~{" THEN
        DO:
            jsonParser  = NEW ObjectModelParser().
            jsonObject  = CAST(jsonParser:Parse(filter), jsonObject).
            ablFilter   = jsonObject:GetCharacter("ablFilter") NO-ERROR.
            cWhere      = "WHERE " + ablFilter.
        END.
        ELSE IF filter NE "" THEN
        DO:
            /* Use filter as WHERE clause */
            cWhere = "WHERE " + filter.
        END.

        CREATE QUERY qh.
        qh:SET-BUFFERS(BUFFER Customer:HANDLE).
        qh:QUERY-PREPARE("PRESELECT EACH Customer " + cWhere).
        qh:QUERY-OPEN ().
        numRecs = qh:NUM-RESULTS.

    END METHOD.

END CLASS.
```

Key changes to note in `Customer.cls` include the following:

- **Added statement: `USING Progress.Json.ObjectModel.*.`** — Supports access to the ABL built-in classes for parsing the JSON Filter Pattern object returned in the `filter` parameter of the `ReadCustomer( )` method.
- **Added `@openapi.openedge.method.property` annotations: `(name="mappingType", value="JFP")` and `(name="capabilities", value="ablFilter,top,skip,id,orderBy")`** — Causes the JSDO created from this Business Entity to translate the Kendo UI DataSource inputs to the `filter` parameter of the `ReadCustomer( )` method to a JSON Filter Pattern object. Without this annotation, this `filter` parameter passes a JSON duplicate of the Kendo UI settings most recently provided by the `filter` configuration property or `filter( )` method on the DataSource.
- **Updated statement in the `ReadCustomer( )` method: `IF filter BEGINS "~{" THEN ... ELSE ...`** — If the `filter` parameter value starts with a left brace, invokes an added method (`JFPFillMethod( )`) to handle an anticipated JSON Filter Pattern; otherwise, the `BATCH-SIZE` attribute on the buffer handle for `ttCustomer` is set to return all records in the result set and the `filter` parameter is passed to the `ReadData( )` method of the inherited `OpenEdge.BusinessLogic.BusinessEntity` abstract class to handle the `filter` string format. (Note that `JFPFillMethod( )` also sets different values for `BATCH-SIZE` based on the `filter` settings.)
- **Added method: `JFPFillMethod( )`** — Parses the property values from the JSON Filter Pattern passed to the `filter` parameter, assigning any that are found to corresponding ABL variables. Any of these variables that contain appropriate values are then used to implement the filtering, sorting, and paging options that are specified. These values determine the `BATCH-SIZE` to return in the `ttCustomer` temp-table for a successful result. Thus, a successful result returns either a single record identified by `id`, a specified page of records (`iMaxRows > 0`), or the entire result set of records in the `ttCustomer` temp-table of the `DATASET dsCustomer` parameter passed as output from the `ReadCustomer( )` method. The record, or set of records, returned represents the result from the specified filtering, sorting, and paging options, if any. Note that an ABL query is used for some options, while the `FILL( )` method

on `dsCustomer` is used for others to copy `Customer` data to `ttCustomer` and update the corresponding `id` and `seq` fields.

- **Added callback method: `AddIdField( )`** — With this callback registered by the `JFPFillMethod( )` in response to the `AFTER-ROW-FILL` event on `dsCustomer`, this method assigns the current sequence number (`seq`) and `ROWID` value (`id`) of the corresponding database record whose `Customer` fields have just been copied (using `FILL( )`) into the corresponding fields of the current `ttCustomer` record.
- **Method: `count( )`, added as a Mobile invoke operation to return the total number of records in the server result set** — Executed as part of returning a server page to the Kendo UI DataSource, this method identifies any `WHERE` string in the `filter` parameter and adds it to a `PRESELECT` query on the target database table that it constructs and opens. It then passes `NUM-RESULTS` on the opened query as the value of its output parameter to provide the total number of records to Kendo UI. Note the URI that you must specify as part of the invoke operation annotations in Developer Studio: `URI="/count?filter=~{filter~}"`.

Note: If you do not add a method like this to the Business Entity and annotate it (in Developer Studio) as an invoke operation, the JSDO throws an exception when the Kendo UI DataSource tries to reference the method as part of reading a server page.

JSDO dialect of the Kendo UI DataSource

To provide data to a Kendo UI widget, such as a Kendo UI Grid, you use a Kendo UI DataSource. This is a JavaScript class (`kendo.data.DataSource`) that allows you to define an instance to provide either local data values or a remote data service of some type that you define using various configuration properties. The DataSource configuration for a remote data service primarily defines the transport options for reading and modifying the data and the schema of the data. For more information, see the Kendo UI documentation on the kendo.data.DataSource.

Although you can define a DataSource to access virtually any remote data service, the Telerik Kendo UI and Backend Services internally support different *dialects* of the Kendo UI DataSource, each of which supports specialized transport and schema definitions for accessing a given remote data service. Thus, the JavaScript Data Object dialect of the Kendo UI DataSource supports access to the JSDO as a remote data service. In doing so, it takes advantage of the built-in data management features of the JSDO and its schema as defined by a JSDO catalog. The JSDO dialect of the Kendo UI DataSource therefore serves a similar function on the Telerik Platform as JSDO Services do in the Mobile App Builder in previous OpenEdge releases.

The JSDO dialect of the Kendo UI DataSource supports:

- **The Kendo UI Grid and other Kendo UI widgets** in much the same way as other dialects and custom instantiations of the Kendo UI DataSource. However, some standard DataSource property and method behavior is implemented based on a transport that defines a JSDO instance as its remote data service.
- **Access to a single-table JSDO instance or any single table from a multi-table JSDO instance**—using a separate DataSource for each table of a shared multi-table JSDO to show master-detail relationships.
- **JSDO dialect transport properties: `jsdo`, `tableRef`, and `countFnName`**, which specify, respectively, the JSDO configured for the DataSource, the name of the JSDO table that the DataSource accesses (optional for single-table JSDOs), the name of the JSDO invoke method that the DataSource must call to return the total number of records in the server result set for the JSDO table (required only for server paging).
- **Pre-configured DataSource CRUD operations** using the individual JSDO CRUD or submit operations already defined for the JSDO instance—there is no need to configure each DataSource CRUD operation individually.
- **Initial values for create operations**, as defined for the JSDO instance.
- **The option to configure a DataSource to instantiate its own JSDO** for private, single-table access, as well as to share an existing JSDO instance.
- **Access to the JSDO instance directly** in DataSource event handlers or Promise deferred functions, for example, to call `addRecords( )`, the local storage APIs, or invoke methods on the JSDO.
- **The standard `sync( )` method** on the DataSource, which calls the JSDO `saveChanges( )` method, depending on a user's interaction with the connected Kendo UI widget.
- **The standard `batch` configuration property** on the DataSource, which indicates if the JSDO calls `saveChanges( )` to invoke a submit operation (if available) to send a group of changes across the network or to invoke each Mobile create, update, and delete operation individually across the network.
- **The standard `serverPaging`, `serverFiltering`, and `serverSorting` configuration properties** of the DataSource, which allow a remote data service to manage the paging, filtering, and sorting features (respectively) of Kendo UI. If any or all of these server properties are set to `true`, the JSDO passes the corresponding property settings for these features to the Mobile read operation to be managed on the server. If any or all of these server properties are set to `false` (the default), the DataSource manages the corresponding features in its own local memory using the data that has already been returned by the Mobile read operation.
- **A generic JSDO server error message** for any single Mobile create, update, delete, or submit operation that returns a server error, allowing you to inspect the XHR object for details of the error or errors (for a submit operation) returned.
- **The `progress.data.JSDOSession` class (described later) to create a JSDO login session** for the DataSource that is validated with either Anonymous, Basic or Form-based authentication over HTTP or HTTPS.

Following is a sample Kendo UI DataSource instantiation that shows specific configuration features of the JSDO dialect, shown in bold:

```
myjsdo = new progress.data.JSDO({ name: 'dsCustomerOrd' });

var dataSource = new kendo.data.DataSource( {
    type: "jsdo",
    serverPaging: true,
    serverFiltering: true,
    filter: { field: "State", operator: "eq", value: "MA" },
    serverSorting: true,
    sort: { field: "State", dir: "desc" },
    transport: {
        jsdo: myjsdo,
        tableRef: "ttCustomer",
        countFnName: "count",
    },
    /* batch: true, */
    error: function(e) {
        var obj, dsErrors, error = "", i;

        console.log('Error: ', e);
        if (e.errorThrown) {
            error = "\n" + e.errorThrown.message;
        }
        if (e.xhr && e.xhr.response) {
            try {
                obj = JSON.parse(e.xhr.response);
                dsErrors = obj.dsCustomerOrd["prods:errors"];

                if (obj._retVal) {
                    error += "\n" + obj._retVal;
                } else if (obj._errors instanceof Array &&
                    obj._errors.length > 0) {
                    for (i = 0; i < obj._errors.length; i += 1) {
                        error += "\n" + obj._errors[i]._errorNum
                                 + " " + obj._errors[i]._errorMsg;
                    }
                }
                else if (dsErrors &&
                         dsErrors.ttCustomer instanceof Array) {
                    for (i = 0; i < obj.dsErrors.ttCustomer.length; i += 1)

                    {
                        error += "\n" +
                            dsErrors.ttCustomer[i]["prods:error"];
                    }
                }
            }
            catch(e) {
                alert("Error while parsing response: " + e.xhr.response);
            }
        }
        alert("Error returned from server: " + error);
        e.preventDefault();
    }
} );
```

Note that this example DataSource accesses a JSDO that is instantiated prior to instantiating the DataSource itself, which also assumes that a JSDO login session has been created and a JSDO catalog for a multi-table ProDataSet has been added to the session for the JSDO. OpenEdge supports a new `progress.data.JSDOSession` class to manage JSDO sessions for use with the Kendo UI DataSource. For more information, see Session management updates on page 31.

Following is a brief description of the configuration options specified in the sample JSDO dialect of the Kendo UI DataSource shown above:

- **Setting the standard `type` property to `"jsdo"`** — Specifies that the Kendo UI DataSource is being instantiated for the JSDO dialect.
- **Setting the standard `serverPaging`, `serverFiltering`, or `serverSorting` properties to `true`** — When any of these properties is set to `true`, the respective DataSource feature is handled by Business Entity resource accessed by the JSDO when the Mobile read operation is invoked. For example, if `serverFiltering` is `true`, the settings from the `filter` property on the DataSource (or from any more recent invocation of the `filter( )` method on the DataSource) are included in the JSON Filter Pattern that is passed to the `filter` parameter of the Mobile read operation. If `serverFiltering` is false, all filtering of data is handled by the DataSource in its internal memory. The same is true of the remaining `server*` configuration properties and their related property settings, like the `pageSize` property for `serverPaging` and the `sort` property for `serverSorting`.

  If none of these properties are `true`, the JSDO returns all records, for the single (or specified) table of the JSDO to the DataSource, which manages any paging, filtering, and sorting using its own internal memory.

  Note: The JSDO dialect of the DataSource does **not** support the `serverGrouping` and `serverAggregates` configuration properties in this release. That is, any data grouping and aggregate features of Kendo UI are managed in Kendo UI internal memory.

- **Setting the JSDO dialect `jsdo` transport property** — The `jsdo` property specifies the JSDO that is accessed by the DataSource. In the sample, an existing instance of the JSDO (`myjsdo`) is specified. You can also specify the name of a JSDO resource as a string, and the DataSource will create the JSDO instance for you. This is the same resource name you would use to create the JSDO yourself. In this sample, you would specify the `jsdo` value as `"dsCustomerOrd"`.
- **Setting the JSDO dialect `tableRef` transport property** — Required only for a JSDO created for a resource that is a multi-table ProDataSet, the `tableRef` property specifies the name of the JSDO table whose data the DataSource handles. So, for an OpenEdge ProDataSet, you specify the name of the corresponding temp-table in the Business Entity (`ttCustomer` in the sample). If you want to access the data in all tables in the ProDataSet, you need to create a separate Kendo UI DataSource for each table accessed by the JSDO and attach it a Kendo UI widget appropriately.
- **Setting the JSDO dialect `countFnName` transport property** — The `countFnName` property specifies the name of the Business Entity invoke operation method that returns the total number of records in the server result to the Kendo UI DataSource when using server paging (`serverPaging == true`). For more information, see Updating Business Entities for new Mobile Apps in the Telerik Platform on page 18. Note that for a JSDO that reads data from a multi-table ProDataSet, this invoke method only returns the total number of records in the result set (in all pages) of the parent table.
- **Setting the standard `batch` property** — Specifies how record changes are processed when the `sync( )` method is called on the DataSource, depending on the `batch` property setting (`false` by default). For this release of the JSDO dialect of the DataSource, if `batch == true`, the DataSource's JSDO **must** define a Mobile submit operation. The following table shows how the DataSource responds to the setting of its `batch` property, depending on whether a submit operation is defined in the JSDO:

Table 2: JSDO dialect response to the batch property setting on the DataSource

DataSource `batch` property setting	Does JSDO define a submit operation?	DataSource response when saving changes
`false`	No	The transport gets a change for a single record. This record change is sent to the server one record at a time, using the corresponding Mobile create, update, or delete operation (by calling `saveChanges(false)` on the JSDO).
`true`	No	Not supported in this release. The DataSource throws an exception if saving changes to the server with `batch == true` when the JSDO does **not** define a submit operation.
`false`	Yes	The transport gets a change for a single record. This record change is sent to the server one record at a time, using the Mobile submit operation (by calling `saveChanges(true)` on the JSDO).
`true`	Yes	The transport gets changes for one or more records in an array. All the changes are grouped by the JSDO and sent to the server using the Mobile submit operation (by calling `saveChanges(true)` on the JSDO).

Note: For a standard Kendo UI DataSource with `batch == true`, multiple record changes are grouped together so that all record changes of a given type (create, update, or destroy) are sent to the server at one time by the DataSource transport. However in this release, the transport for the JSDO dialect can **only** send the entire set of record changes (all create, update, and destroy changes together) in a single Mobile submit operation.

- **Handling the parameter (`e`) returned by the DataSource `error` event handler function** — The `error` event handler function is called when an error occurs while performing CRUD operations in the remote data service (JSDO in this case). An error can thus be returned when invoking the `read( )` method or the `sync( )` method on the DataSource.

  The `errorThrown`, `sender`, `status`, and `xhr` properties of the event parameter (`e`) are set depending on the error returned. To process these errors, you can look at `e.errorThrown.message` and at the XmlHttpRequest (XHR) response for more information. The XHR response can have one of the following formats:

  - An error message object
  - An error message text
  - A dataset structure with errors in the `prods:errors` property

Note that if an error occurs when invoking the DataSource `sync( )` method when saving changes in the JSDO to the server, the following generic message is typically returned in `e.errorThrown.message`: `"JSDO: Error while saving changes."`. This message is returned if one or more errors occurs while saving JSDO changes, regardless of the `batch` property setting on the DataSource. So for example, if `batch == true`, and multiple errors occur for a single Mobile submit operation, this one error message is returned for all of them.

For a JSDO dialect `read( )` or `sync( )` call, you can explore the XHR response from the server for more information about any of these errors. There are two basic approaches you can use, when:

- Mobile create, read, update, and delete (CRUD) operations called on the JSDO process data **without** before-image support.
- Mobile CRUD and submit operations called on the JSDO process data **with** before-image support. In this case, error information can be included in one or more changed record objects that are returned for the request.

Note: Submit operations always process data with before-image support. Note also that even when the DataSource invokes `sync( )` for one record change operation at a time (with `batch == false`), it invokes a Mobile submit operation (if it is defined in the JSDO) for each record.

The error handler in the sample follows one or the other of these two approaches, as only one can occur for each DataSource request.

To locate and handle the errors for individual Mobile CRUD operations **without** before-image support, this XHR response contains any error information that is returned in the `response` property object returned in the JSDO request object from the JSDO `saveChanges( )` operation (see the OpenEdge JSDO reference documentation for more information). The example `error` event handler function shows one example of how you can inspect the XHR response to handle error information returned in the `response` property object of the XHR, by accessing the `_retVal` and `_errors` properties of this object.

To locate and handle the errors for individual Mobile CRUD and submit operations **with** before-image support, you have to access a `prods:errors` property in the dataset structure (`dsCustomerOrd`) found in the same `response` property object. This property contains any application error information returned by the Business Entity for each temp-table record where a record change operation returned an error. The application error associated with each such record in this example is in `dsCustomerOrd["prods:errors"].ttCustomer[i]["prods:error"]`. Note that for multiple record change errors that are returned for a submit operation, the example shows how you can return them from the corresponding table array (`ttCustomer[i]["prods:error"]`). For individual CUD operations working with similar before-image data, there can be only one table record that provides this error information for each Mobile CUD operation. A Mobile read operation, of course, might also return errors in multiple table records.

In both cases, the sample concatenates error information at each level of detail for a given server request.

Session management updates

This section describes session management updates to support new OpenEdge Mobile App development using the Telerik Platform and includes an additional note about testing OpenEdge Mobile Apps that use HTTPS on mobile devices.

The main update is a new `progress.data.JSDOSession` class that now supports the recommended approach to session management for JSDO access to Mobile resources, and is particularly suited for use with the JSDO dialect of the Kendo UI DataSource. The primary feature of this class (as distinguished from the previously released `progress.data.Session` class) is that all of the session management methods that access the server execute asynchronously only and return a jQuery implementation of the Promise instead of firing event callbacks to handle results. This support applies to the following methods:

- `addCatalog( )`
- `login( )`
- `logout( )`
- `ping( )`

A Promise is a deferred object whose methods execute when results from the method that returns the Promise are available. You provide your own code to handle the results returned by a given Promise method, much like the code you provide in an event callback. The primary Promise methods for use with a `JSDOSession` object include `done( )`, `fail( )`, and `always( )`, which allow you to handle the successful, unsuccessful, and all results (respectively) of a `JSDOSession` method execution. The description of the `JSDOSession` class and its methods and properties show several examples using Promises. For more information on the jQuery implementation of Promises and the additional Promise methods that are available, see the jQuery topics at the following Web locations:

- **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
- **Promise Object** — http://api.jquery.com/Types/#Promise
- **.promise** — http://api.jquery.com/promise/

Another distinguishing feature of a `JSDOSession` object, is that you instantiate the object and invoke the `login( )` and `addCatalog( )` methods with different calling sequences than for a `Session` object. These differences support a more flexible session management model that better supports subsequent and different user session logins and logouts from the same Mobile Web application using the same `JSDOSession` object. You can also load multiple JSDO catalogs more easily and without first logging the`JSDOSession` object into a Mobile Web application.

For more information on these updates, see:

- progress.data.JSDOSession class on page 32
- JSDOSession methods on page 38
- JSDOSession properties on page 53
- JSDOSession events on page 61
- Note about certificate management for HTTPS testing on page 65

progress.data.JSDOSession class

The `progress.data.JSDOSession` is a JavaScript class that provides methods, properties, and events to manage user access to remote Mobile resources using `progress.data.JSDO` class instances (JSDOs). This includes loading the definitions for the Mobile resources that JSDOs can access and starting and managing user login sessions on Web servers that provide access to these Mobile resources.

This `JSDOSession` class supports similar features to the `progress.data.Session` class, except that `JSDOSession` methods for loading Mobile resource definitions and managing login sessions **only** run asynchronously and return jQuery Promises to handle the results, where as the equivalent `progress.data.Session` methods either run synchronously or fire asynchronous events to allow results to be handled in separately registered callbacks. In addition, the information required to instantiate and start user login sessions with `JSDOSession` objects is handled somewhat differently in order to support a more flexible session management model, and the process of loading Mobile resource definitions into a `JSDOSession` object does not require the object to start a login session as a prerequisite.

Like a `Session` object, a `JSDOSession` object manages user authentication and session identification information in HTTP/S messages sent between JSDOs running in a Mobile App and Mobile services running on a Web server, each of which provide access to a set of Mobile resources. The authentication information includes a user ID and password (*user credentials*), if necessary, to authenticate a user login session in a Mobile Web application, which provides a REST transport between a set of Mobile services and any client Mobile App that accesses them. The session identification information includes the URI for the Mobile Web application and a client context ID that identifies the user login session and helps to manage interactions between the client and the Mobile Web application.

To start a user login session for a `JSDOSession` object to manage, you first instantiate the `JSDOSession` for a particular Mobile Web application and its authentication model, and then invoke the object's `login( )` method. You pass the `login( )` method any required user credentials to authenticate `JSDOSession` access to the Mobile Web application according to its authentication model. Once a `JSDOSession` object is logged in to a Mobile Web application, the Web application provides client access to all Mobile services that the Web application defines, and each of its Mobile services provides access to a defined set of one or more Mobile resources.

Each Mobile service provides a separate JSON file (JSDO catalog) that defines the schema for its set of Mobile resources and the Mobile operations to communicate between these Mobile resources and the JSDO instances that access them from a Mobile App. To create a JSDO instance to access a Mobile resource, a `JSDOSession` object must load the catalog that defines the resource using its `addCatalog( )` method either before or after it logs into the Mobile Web application. This method can load this catalog from the Mobile Web application, from some other Web location, or from a location on the client where the Mobile App runs, and the method can also accept credentials to access a protected JSDO catalog prior to logging into the Mobile Web application or to access a JSDO catalog that is protected separately from the Mobile Web application itself.

Once a JSDO catalog is loaded into a `JSDOSession` object, you can instantiate a JSDO to access a particular Mobile resource defined in that catalog. Once the `JSDOSession` is logged into its Mobile Web application, the JSDO can then access its Mobile resource, relying on any authentication information for the login session (if necessary) to authorize this access.

Multiple JSDOs can thus rely on a single `JSDOSession` object to manage session access to all Mobile services and their resources defined by the same Mobile Web application.

Constructor

Instantiates a `JSDOSession` object that you can use to start and manage a user login session in a Mobile Web application and load the JSDO catalog for each supported Mobile service whose resources are accessed using JSDOs.

Syntax

```
progress.data.JSDOSession ( options )
```

options

An object that contains the following configuration properties:

- **`serviceURI`** — (Required) A string expression containing the URI of the Mobile Web application for which to start the user login session. This Mobile Web application must support one or more OpenEdge Mobile services in order to create JSDOs for the service resources provided by the application. This URI is appended with a string that identifies a resource to access as part of the login process.

  If the Mobile App from which you are logging in is a Mobile Web App deployed to the same Apache Tomcat server as the Mobile Web application, you can specify `serviceURI` as a relative URI, for example, `/SportsMobileApp`, which is relative to the deployment end point (Tomcat server domain or host and port).

  If the Mobile App from which you are logging in is a Mobile Native App that will be installed to run directly in a native device container, or if it is a Mobile Web App deployed to a different Web server from the Mobile Web application, you must specify `serviceURI` as an absolute URI to the Tomcat server domain or host and port, for example, `http://www.progress.com/SportsMobileApp`, or perhaps for testing, `http://testmach:8980/SportsMobileApp`.

  Note: Once the `login( )` method executes, the value you pass for `serviceURI` also sets the value of the `sessionURI` property on the current `JSDOSession` object, whether or not user login completes successfully.

- **`authenticationModel`** — (Optional) A string constant that specifies one of the three authentication models that the `JSDOSession` object supports:

  - **`progress.data.Session.AUTH_TYPE_ANON`** — The Mobile Web application supports Anonymous access. No authentication is required. This is the default value if you do not specify a value for this property.
  - **`progress.data.Session.AUTH_TYPE_BASIC`** — The Mobile Web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the Web application's resources for you, you need to pass these credentials in a call to the `JSDOSession` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your Mobile App, either using a form of your own design or using a template provided by Progress Software Corp.
  - **`progress.data.Session.AUTH_TYPE_FORM`** — The Mobile Web application uses Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the Web application itself sends a form to the client to get the credentials. However,

when you have the `JSDOSession` object manage access to the Web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `JSDOSession` intercepts the form sent by the Web application and handles the authentication without that form being displayed.

If the Mobile Web application requires authentication, you must set this value correctly to ensure that users can log in.

Properties

Table 3: progress.data.JSDOSession properties

Member	Brief description (See also the reference entry)
authenticationModel property (JSDOSession class) on page 53	Returns a string constant that specifies the authentication model that the current `JSDOSession` object is using and was passed as an option to the object's class constructor.
catalogURIs property on page 53	Returns the list of URIs successfully used to load JSDO catalogs into the current `JSDOSession` or `Session` object.
clientContextId property on page 54	The value of the most recent client context identifier (CCID) that the current `JSDOSession` or `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.
connected property on page 54	Returns a `Boolean` that indicates the most recent online state of the current `JSDOSession` or `Session` object, when it last determined if the Mobile Web application it manages is available.
JSDOs property on page 55	Returns an array of JSDOs that use the current `JSDOSession` or `Session` object to communicate with their mobile services.
loginHttpStatus property on page 55	Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` or `Session` object.
loginResult property on page 56	Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` or `Session` object.
onOpenRequest property on page 56	Returns the reference to a user-defined callback function that the `JSDOSession` or `Session` object executes to modify a request object before sending the request object to the server.

Member	Brief description (See also the reference entry)
pingInterval property on page 58	A `Number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` or `Session` object's `ping( )` method and the next.
services property on page 59	Returns an array of objects that identifies the Mobile services that have been loaded for the current `JSDOSession` or `Session` object and its Mobile Web application.
serviceURI property on page 60	Returns the URI to the Mobile Web application that has been passed as an option to the class constructor for the current `JSDOSession` object or that has been passed as a parameter to the most recent call to the `login( )` method on the current `Session` object, whether or not the most recent call to `login( )` succeeded.
userName property on page 61	Returns the user ID passed as a parameter to the most recent call to the `login( )` method on the current `JSDOSession` or `Session` object.

Methods

Table 4: progress.data.JSDOSession class-instance methods

Member	Brief description (See also the reference entry)
addCatalog( ) method (JSDOSession class) on page 38	Loads one or more local or remote JSDO catalogs into the current `JSDOSession` object.
login( ) method (JSDOSession class) on page 41	Starts a user login session in a Mobile Web application for the current `JSDOSession` object by sending an HTTP request with user credentials to the Web application URI specified in the object's constructor.
logout( ) method (JSDOSession class) on page 45	Terminates the login session on the Mobile Web application managed by the current `JSDOSession` object, and reinitializes most of the state information maintained by the object.
ping( ) method (JSDOSession class) on page 47	Determines the online state of the current `JSDOSession` object from its ability to access the Mobile Web application that it manages, and for an OpenEdge Web application, from detecting if its associated application server is running.

Member	Brief description (See also the reference entry)
subscribe( ) method (JSDOSession class) on page 50	Subscribes a given event handler function to an event of the current `JSDOSession` object.
unsubscribe( ) method (JSDOSession class) on page 51	Unsubscribes a given event handler function from an event of the current `JSDOSession` object.
unsubscribeAll( ) method on page 52	Unsubscribes all event handler functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event handler functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Events

Table 5: progress.data.JSDOSession events

Member	Brief description (See also the reference entry)
offline event on page 61	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone offline, or that the Mobile Web application to which it has been connected is no longer available.
online event on page 63	Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone online after it was previously offline, or that the Mobile Web application to which it is connected is now available after it was previously unavailable.

Example — Using the OpenEdge JSDOSession class

This is an example of how you might create a `JSDOSession` object using the URI and authentication model of a Mobile Web application (`CustService`), log into the Web application, load the JSDO catalog for a Mobile service provided by that Web application, and create a Kendo UI Grid with a JSDO dialect of the Kendo UI DataSource that creates a JSDO for a `Customer` Mobile resource defined by that Mobile service in the loaded catalog:

```
var session;

/* create jsdoSession   */
session = new progress.data.JSDOSession(
        { serviceURI: "http://localhost/CustService",
          authenticationModel: progress.data.Session.AUTH_TYPE_FORM
        };

/* create login screen (using UI defined in HTML file) */
window.loginView = kendo.observable( {
    submit: function() {
        var loginParams = {
```

```
                    username: this.username,
                    password: this.password) };

        /* log in (also loads CustService.json if it succeeds) */
        session.login(loginParams
            ).done( // Logged in
            function(session, result, info) {
                session.addCatalog("http://localhost/.../CustService.json"
                    ).done( // JSDO catalog loaded
                    function(session, result, details){
                        /* success function - create grid and datasource */
                        $("#grid").kendoGrid( {
                            dataSource: {
                                type: "jsdo",
                                transport: {
                                    jsdo: { resourceName: "Customer" }
                                },
                                error: function(e) {
                                    console.log("Error: ", e);
                                }
                            },
                            /* etc., other kendoGrid properties */
                        };

                        /* switch UI to show grid */
                        window.location.href = "#grid";
                    }).fail( // JSDO catalog not loaded
                    function(session, result, details){
                        alert("JSDO catalog failed to load");
                }); //JSDOSession addCatalog()

            }).fail( // Not logged in
            function(session, result, info) {
                /* display error message, stay on login screen */
                alert("Login failed");
        }); // JSDOSession login()
    } // observable submit method
} ); // kendo.observable
```

The JSDO automatically finds and uses the `JSDOSession` object on which a catalog that defines the `Customer` resource is loaded.

Notes

- The four Web-communicating methods of this class, `login( )`, `addCatalog( )`, `logout( )`, and `ping( )`, all execute asynchronously, and return a jQuery Promise as the result. A Promise is a deferred object whose methods execute when results from the method that returns the Promise are available. You provide your own code to handle the results returned by a given Promise method, much like the code you provide in an event callback. The primary Promise methods for use with a `JSDOSession` object include `done( )`, `fail( )`, and `always( )`, which allow you to separately handle the successful, unsuccessful, and all results (respectively) of a `JSDOSession` method execution. The parameter list for all of these primary Promise methods is the same for a given `JSDOSession` method, but each Promise method returns a slightly different or identical set of parameter values, depending on its function. For example, the `always( )` method executes with values in its parameter list that are identical to the values passed to either the `done( )` or `fail( )` method, whichever one executes with results.

  For more information on the jQuery implementation of Promises and the additional Promise methods that are available, see the jQuery topics at the following Web locations:

  - **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
  - **Promise Object** — http://api.jquery.com/Types/#Promise

- **.promise** — http://api.jquery.com/promise/

- Use an instance of this class to call the `login( )` method to start a user login session, call the `addCatalog( )` method to load one or more JSDO catalogs for the session, and possibly call the `logout( )` method to terminate the session. To use the same `JSDOSession` instance to start a new login session, you must call the `logout( )` method first. Call the `ping( )` method to determine if the `JSDOSession` object is in an online state, able to access its Mobile Web application and associated application server.
- The behavior of a login session using this class depends on the authentication model of the Web server and how its resources are protected and accessed. For more information, see the description of the `login( )` method.
- If you have special requirements for sending `JSDOSession` requests to the Web server, you can modify the XMLHttpRequest object that is sent by the `JSDOSession` object. To do so, assign a callback function as the value of `onOpenRequest` property.

See also

addCatalog( ) method (JSDOSession class) on page 38, login( ) method (JSDOSession class) on page 41, logout( ) method (JSDOSession class) on page 45, `progress.data.JSDO` class

JSDOSession methods

The following sections describe the methods of the `progress.data.JSDOSession` class.

addCatalog( ) method (JSDOSession class)

Loads one or more local or remote JSDO catalogs into the current `JSDOSession` object.

The appropriate catalog must be loaded before creating a JSDO for any resource defined in the catalog.

If a catalog is on a Web server (remote), this method throws an exception if it is not possible to send a request to the specified Web application.

This method is always called asynchronously and can be called either before or after the `JSDOSession` object has successfully established a login session using its `login( )` method. If a JSDO catalog is stored remotely in the Mobile Web application, the recommended practice is to log into the Web application before calling this method.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 32

Syntax

```
addCatalog ( catalog-uri [ , cat-user-name , cat-password ] )
addCatalog ( catalog-uris )
```

catalog-uri

The URI of a JSDO catalog file, which can specify either a location (remote) in a Web application running on a Web server or a location (local) that is relative to where the

App is currently running. If the URI is a relative path (e.g., `catalogs/OrderEntrySvc.json`), the *`catalog-uri`* is assumed to be relative to the location from which the App was loaded.

If the `JSDOSession` object has already logged in and the Mobile App is Web App deployed to the same Apache Tomcat server as the Mobile Web application, you can specify a URI that is relative to the deployment end point (Tomcat server domain or host and port), for example: `/SportsMobileApp/static/mobile/OrderEntrySvc.json`, where `/SportsMobileApp` is the location of the Web application.

If the Mobile App from which you are logging in is a Hybrid App that will be installed to run directly in a native device container, or if it is a Mobile Web App deployed to a different Web server from any Web application to which the `JSDOSession` object has already logged in, you must specify an absolute URI that includes the Tomcat server domain or host and port, for example, `http://www.progress.com:8980/SportsMobileApp/static/mobile/OrderEntrySvc.json`, or perhaps for testing, `http://testmach:8980/SportsMobileApp/static/mobile/OrderEntrySvc.json`.

Note: The default catalog URI for a catalog created for an OpenEdge Mobile service, relative to the Apache Tomcat server domain or host and port where the session is logged in, is the following: `/`*`MobileWebApplicationName`*`/static/mobile/`*`ServiceName`*`.json`, where *`MobileWebApplicationName`* is the name of the Mobile Web application and *`ServiceName`* is the name of the Mobile service for which the JSDO catalog is created.

Note: For the *`catalog-uri`* parameter, if the `JSDOSession` object is already logged in, you typically do not need to specify *`cat-user-name`* and *`cat-password`*. These optional parameters are available primarily if the object is not yet logged in, or if you store the catalog somewhere other than in the Web application to which the object is logged in.

`cat-user-name`

A string expression containing a user ID to authenticate access to a protected catalog. If you do not specify *`cat-user-name`*, catalog access is authorized using existing user credentials (if any).

`cat-password`

A string expression containing a password (if required) to authenticate the user specified by *`cat-user-name`*.

`catalog-uris`

An array of strings, each of which is the URI for a catalog as defined for the *`catalogURI`* parameter. All of the catalogs in the array must be accessible without specifying credentials. This can be either because they are unprotected, or because the `JSDOSession` object has already logged into the Web application where they are located.

Note: You can read the `catalogURIs` property to return the URIs for all catalogs previously loaded into the `JSDOSession` object.

Promise method signatures

Following are the signatures of methods that are observed in the jQuery Promise that `addCatalog( )` returns:

Syntax:

```
done: function ( session , result , details )
fail: function ( session , result , details )
always: function ( session , result , details )
```

session

A reference to the `JSDOSession` object on which `addCatalog( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **progress.data.Session.SUCCESS** — Each catalog specified by the *catalogURI* or *catalogURIs* parameter of `addCatalog( )` has been loaded successfully or has already been loaded.
- **progress.data.Session.GENERAL_FAILURE** — One or more of the specified catalogs has failed to load successfully.

details

An array of JavaScript objects that contain information on the JSDO catalogs that `addCatalog( )` attempted to load. Each object has the following properties:

- **catalogURI** — The URI of a specified catalog.
- **result** — A constant indicating the result of loading the catalog that can have one of the following values:
  - **progress.data.Session.SUCCESS** — The specified catalog loaded successfully, or has previously been loaded.
  - **progress.data.Session.CATALOG_ALREADY_LOADED** — The specified catalog was previously. loaded.
  - **progress.data.Session.AUTHENTICATION_FAILURE** — The specified catalog failed to load because of an authentication error.
  - **progress.data.Session.GENERAL_FAILURE** — The specified catalog failed to load because of an error other than an authentication failure.
- **errorObject** — Any error object thrown while attempting to load the catalog.
- **xhr** — A reference to the XMLHttpRequest object used to load the catalog from a Web server.

Note: When `addCatalog( )` is requested to load a JSDO catalog that is already loaded, it does not load the catalog again. This is considered to be a successful execution. That is, the `done( )` method is called, although the `result` value in the *details* object for the catalog is `progress.data.Session.CATALOG_ALREADY_LOADED`.

Example

The following code fragment calls the `addCatalog(myCatalogURIs)` method on the `JSDOSession` object, `mySession`, to load multiple JSDO catalogs specified in `myCatalogURIs`. This example uses `try` and `catch` blocks to check any error object thrown prior to requesting the catalog, and assembles an appropriate message to display in an alert box or does other processing for each case:

```
try {
    mySession.addCatalog( myCatalogURIs ).done(
        function( session, result, details ) {
            alert("All catalogs loaded.");
        }).fail(
        function( session, result, details ) {
            var numCats = details.length;
            for ( i = 0; i < numCats; i++ ) {
                if (details[i].result
                    === (progress.data.Session.SUCCESS
                         || progress.data.Session.CATALOG_ALREADY_LOADED) {
                    alert("Catalog successfully loaded: " +
details[i].catalogURI);
                } else if (details[i].result
                    === progress.data.Session.AUTHENTICATION_FAILURE) {
                    alert("Authentication error: " + details[i].catalogURI);

                } else if (details[i].result
                         === progress.data.Session.GENERAL_FAILURE) {
                    alert("General catalog load error: "
                          + details[i].catalogURI);
                    if (details[i].errorObject) {
                        // Process thrown error object during load . . .
                    }
                    if (details[i].xhr) {
                        // Process XHR object sent for the load . . .
                    }
                } else {
                    alert("Not sure what is wrong with "
                          + details[i].catalogURI);
                }
            } // for each load attempt
        });
}
catch(errObj) {
    var msg;

    msg = '\n' + errObj.message;
    alert("Unexpected addCatalog() error: " + msg);
}
```

See also:

catalogURIs property on page 53, login( ) method (JSDOSession class) on page 41

login( ) method (JSDOSession class)

Starts a user login session in a Mobile Web application for the current `JSDOSession` object by sending an HTTP request with user credentials to the Web application URI specified in the object's constructor.

This method throws an exception if it is not possible to send a request to the specified Web application.

On a successful login, the `JSDOSession` object sets its `connected` property to `true`. If the login fails, the object leaves its `connected` property set to `false`.

This method is always executed asynchronously.

Note: Before invoking this method, ensure that you set the `authenticationModel` configuration property in the constructor of the `JSDOSession` object correctly (see the notes on authentication models).

Note: If the browser or mobile device has already authenticated a user login session, this method completes successfully.

Note: This method does not support proxy servers (servers that function as a security service).

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 32

Syntax

```
login ( [ username , password ] )
```

`username`

A string expression containing a user ID for the method to send to the Web server for authentication.

Note: The `userName` property of the `JSDOSession` object returns the most recent value passed to this method for the current `JSDOSession` object.

`password`

A string expression containing a password for the method to send to the Web server to authenticate the specified user.

Promise method signatures

Following are the signatures of methods that are observed in the jQuery Promise that `login( )` returns:

Syntax:

```
done: function ( session , result , info )
fail: function ( session , result , info )
always: function ( session , result , info )
```

`session`

A reference to the `JSDOSession` object on which `login( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **progress.data.Session.AUTHENTICATION_SUCCESS** — The user login session started successfully. You can use JSDOs to access any Mobile services supported by the Web application to which the `JSDOSession` object has logged in.
- **progress.data.Session.AUTHENTICATION_FAILURE** — User login failed because of invalid user credentials (*username* or *password*).
- **progress.data.Session.GENERAL_FAILURE** — User login failed because of a non-authentication failure.

info

A JavaScript object that can have the following properties:

- **errorObject** — Any error object thrown as a result of sending a login request to the Web server.
- **xhr** — A reference to the XMLHttpRequest object sent to the Web server to start a user login session.

You can also return the result of the most recent login attempt on the current `JSDOSession` object by reading its `loginResult` property. For a more specific status code returned in the HTTP response, you can also check the value of the `loginHttpStatus` property.

Note: You can log out from a Web application and then log in again using the same `JSDOSession` object. The login will use the same `serviceURI` and `authenticationModel` settings originally passed to the constructor, but you must pass any required credentials each time `login( )` is called.

General Web server interaction

The general Web server interaction with and response to this method depends on the authentication model that the Web server uses and how resources are accessed and protected. You configure the authentication model for each Mobile Web application deployed to the Apache Tomcat and specify both the Web application URI and its corresponding authentication model to the `JSDOSession` object constructor. For more information on the authentication models that a `JSDOSession` object supports, see the description of the constructor for the progress.data.JSDOSession class on page 32.

For more information on these authentication models and how to configure them for a Mobile Web application, see the deployment sections on Web server authentication models in *OpenEdge Development: Mobile Applications* and your Web server documentation.

Caution: You must be sure that security is configured to complete authentication before the application requests resources in the JSDO catalog. Although it is possible to configure application security so that only the Mobile resources in the catalog require authentication, Progress Software does not recommend this approach. Instead, Progress Software recommends that you require authentication for application resources in addition to those defined in the catalog, and require that the authentication occur prior to accessing any resources in the catalog. Once the user is authenticated, the Web server provides access to all other resources, including catalog resources, according to the user's authorization settings.

Note: Unless the application design guarantees that the user will be prompted by the Web browser or native device container to provide credentials before a `login( )` call occurs, Progress Software recommends (in some cases requires) that the Mobile App pass the credentials as parameters to the `login( )` method. In addition, you must correctly pass the value of the `authenticationModel` configuration property to the `JSDOSession` object's constructor. Coding the Mobile App in this way ensures that the proper credentials are submitted to the server and promotes a favorable user experience.

Example

The following code fragment calls the `login( )` method on the `JSDOsession` object, `empSession`. This example uses the Promise `done( )` and `fail( )` methods to check the result of the call along with any error object thrown as a result of the request, then assembles an appropriate message to display in an alert box. It also uses a `try-catch` block in case the `login( )` method throws an error object instead of sending the request to the server:

```
var msg;
var xhr;

try {
    empSession.login( { userName : uname,
                        password : pw }).done(
        function( session, result, info ) {
            msg = "Logged in successfully”;
        }).fail(
        function( session, result, info ) {
            if ( result
                 === progress.data.Session.AUTHENTICATION_FAILURE ) {
                msg = "Employee Login failed. Authentication error";
            }
            else if ( result
                      === progress.data.Session.GENERAL_FAILURE  ) {
                msg = "Employee Login failed. Unspecified error";
                if ( info.errorObject ) {
                    // Process error object thrown during login . . .
                }
                if ( info.xhr ) {
                    // Process XHR sent during login . . .
                }
            }
            xhr = info.xhr;
        });
}
catch(errObj) {
    msg = "Employee Login failed. Error attempting to call login";
    msg = msg + '\n' + errObj.message;
}

msg = msg +
          "\nloginResult: " + empSession.loginResult +
          "\nloginHttpStatus: " +  empSession.loginHttpStatus +
          "\nuserName: " +  empSession.userName +
          "\nLogin XHR: " +  xhr;

alert(msg);
```

See also:

addCatalog( ) method (JSDOSession class) on page 38, authenticationModel property (JSDOSession class) on page 53, connected property on page 54, loginHttpStatus property on page 55, loginResult property on page 56, logout( ) method (JSDOSession class) on page 45, offline event on page 61, online event on page 63, serviceURI property on page 60, userName property on page 61

logout( ) method (JSDOSession class)

Terminates the login session on the Mobile Web application managed by the current `JSDOSession` object, and reinitializes most of the state information maintained by the object.

This method throws an exception if it is not possible to send a request to the specified Web application.

On a successful logout, the `JSDOSession` object sets its `connected` property to `false`. If the logout fails, the object leaves its `connected` property set to `true`, unless the failure happened because the App cannot access the server.

Once `logout( )` is executed, no further communication (other than a `login( )` call) can occur between the Mobile App and the server using this `JSDOSession` object. However, any catalogs loaded in the object remain available to create and maintain JSDOs, though these JSDOs cannot make requests to the server until a new login session is established for the object (including for anonymous access).

This method is always executed asynchronously.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 32

Syntax

```
logout ( )
```

Promise method signatures

Following are the signatures of methods that are observed in the jQuery Promise that `logout( )` returns:

Syntax:

```
done: function ( session , result , info )
fail: function ( session , result , info )
always: function ( session , result , info )
```

session

A reference to the `JSDOSession` object on which `logout( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The logout completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — User logout failed because of some error.

`info`

A JavaScript object that can have the following properties:

- **`errorObject`** — Any error object thrown while attempting to terminate the login session.
- **`xhr`** — A reference to the XMLHttpRequest object sent to the Web server to terminate the user login session.

Detailed logout behavior

When this method terminates the associated login session, the `JSDOSession` object can be re-used to start a new session using the same `serviceURI` and `authenticationModel` configuration settings originally passed to the object's constructor. The `JSDOSession` object's properties retain their values from the previous login session, with the following exceptions:

- `clientContextId` is reset to `null`.
- `loginHttpStatus` is reset to `null`.
- `loginResult` is reset to `null`.
- `userName` is reset to `null`.

Existing JSDOs and catalog information are not affected by a successful execution of `logout( )`. However, any attempt to call `addCatalog(` or a JSDO method that requires contacting the server results in an error object being thrown.

Example

The following code fragment calls the `logout( )` method on the `JSDOsession` object, `empSession`. This example uses `try` and `catch` blocks in case `logout( )` throws an unexpected error object, and displays messages accordingly:

```
var msg;

try {
    empSession.logout( ).done(
        function( session, result, info ) {
            msg = "Logged out successfully”;
        }).fail(
        function( session, result, info ) {
            if ( result === progress.data.Session.GENERAL_FAILURE  ) {
                msg = "Employee Logout failed. Unspecified error";
                if ( info.errorObject ) {
                    // Process error object thrown during logout . . .
                }
                if ( info.xhr ) {
                    // Process XHR sent during logout . . .
                }
            else {
                msg = "Unexpected logout result";
                if ( info.errorObject ) {
                    // Process error object thrown during logout . . .
                }
                // Process info.xhr, if necessary, for more information
            }
        });
    alert(msg);
}
catch(errObj) {
    msg = errObj ? '\n' + errObj.message : '';
    alert("There was an unexpected error attempting log out." + msg);
}
```

See also:

login( ) method (JSDOSession class) on page 41, offline event on page 61, online event on page 63

ping( ) method (JSDOSession class)

Determines the online state of the current `JSDOSession` object from its ability to access the Mobile Web application that it manages, and for an OpenEdge Web application, from detecting if its associated application server is running.

The method also causes an `offline` or `online` event to fire if the ping detects that there has been a change in the object's online state.

This method throws an exception if the `JSDOSession` object has not logged into its Mobile Web application or has since logged out.

This method is always executed asynchronously.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 32

Syntax

```
ping ( )
```

Promise method signatures

Following are the signatures of methods that are observed in the jQuery Promise that `ping( )` returns:

Syntax:

```
done: function ( session , result , info )
fail: function ( session , result , info )
always: function ( session , result , info )
```

`session`

A reference to the `JSDOSession` object on which `ping( )` was called.

`result`

A `Boolean` that indicates the online state of the `JSDOSession` object on which `ping( )` is called. If set to `true`, `ping( )` has determined that the current `JSDOSession` object is connected and logged into a Mobile Web application (and its associated application server, for an OpenEdge Mobile service), or if set to `false`, the object is disconnected from any Mobile Web application (**or** its associated application server, for an OpenEdge Mobile service).

`info`

A JavaScript object that can have the following properties:

- **`xhr`** — A reference to the XMLHttpRequest object sent to the Web server to make the ping request to the Web application.
- **`offlineReason`** — A string constant that `ping( )` sets only if it determines that the `JSDOsession` object is disconnected from any Mobile Web application or its associated application server. The constant value indicates the reason for its offline state. Possible values are the same as those that can be passed to the `offline` event handler function as the value of its *`off-line-reason`* parameter, and include:
  - **`progress.data.Session.APPSERVER_OFFLINE`** — (OpenEdge Mobile services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
  - **`progress.data.Session.DEVICE_OFFLINE`** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.

    Note: This condition will fire the `offline` event on a `JSDOSession` or `Session` object even if the object has not yet invoked its `login( )` method.

  - **`progress.data.Session.SERVER_OFFLINE`** — The Web server is not available. For a Rollbase Mobile service, this is the Web server for the public or

private cloud. For an OpenEdge Mobile service, this is the Tomcat Java servlet container.

- **progress.data.Session.WEB_APPLICATION_OFFLINE** — The server is running, but the Java Web application that implements the Mobile service is not deployed.

Detailed ping behavior

The `ping( )` method fires an `online` or `offline` event on its `JSDOSession` object if the method detects that the online state of its `JSDOSession` object has changed. A `JSDOSession` object is considered to be in an online state if all of the following are true:

- The object can communicate with its Mobile Web application. That is, the Web server is running, the Web application is started, and it has accessible Mobile services.
- For OpenEdge Mobile services only, the associated OpenEdge application server that the Web application accesses is running and accessible.

Note: You can have `ping( )` executed automatically by setting the `pingInterval` property on the `JSDOSession` object, which specifies the time interval between invocations of `ping( )`.

Note: For OpenEdge Mobile services, an OpenEdge application server supports `ping( )` using an OpenEdge-defined ABL class, `OpenEdge.Rest.Admin.AppServerStatus`. This class responds to a REST ping service call to its `ServerStatus( )` method, which indicates that the application server is available when the method returns successfully. You can also define a version of this method in your own user-defined ABL class that returns a custom string value when `ping( )` returns successfully, and you can retrieve this value from the `xhr` object reference returned by `ping( )`. For more information, see the sections on constructing and debugging REST requests in the administration documentation for the OpenEdge AppServer and the Pacific Application Server for OpenEdge.

Example

The following code fragment shows how you can use `ping( )` to check the online state of a given `JSDOSession` object after logging in. It also shows that the session `online` and `offline` events can fire both using `ping( )` if the `JSDOSession` object's online state has changed **after** logging in, and without using `ping( )` when the device online status has changed even **prior** to logging in:

```
var mySession = new progress.data.JSDOSession ( { . . . } );

/* These session events can fire on device online
   status changes even prior to logging in */
mySession.subscribe('offline', onSessionOffline );
mySession.subscribe('online', onSessionOnline );

. . .

mySession.login().done( // Anonymous login
    function( session, result, info ) {

        . . .

        ping().done(
            function( session, result, info ) {
                console.log("Session ping result: Online");
                }
            }).fail(
            function( session, result, info ) {
                console.log("Session ping result: Offline -- "
                                + info.offlineReason);
                // Process info.xhr, if necessary, for more information
            });

        . . .

    }).fail( /* Process failed login . . . */ );

. . .

function onSessionOffline( pSession , pOfflineReason , pRequest ) {
  console.log("The session status has changed to: Offline -- "
              + pOfflineReason);
};

function onSessionOnline( pSession , pRequest ) {
  console.log("The session status has changed to: Online");
};
```

See also:

connected property on page 54, offline event on page 61, online event on page 63, pingInterval property on page 58

subscribe( ) method (JSDOSession class)

Subscribes a given event handler function to an event of the current `JSDOSession` object.

This method throws an exception if the specified event is not supported by the `JSDOSession` class.

Return type: `undefined`

Applies to: progress.data.JSDOSession class on page 32

Syntax

```
subscribe ( event-name , event-handler [ , scope ] )
```

event-name

The name of an event on a `JSDOSession` object to which you subscribe an event handler. See the reference entry for the progress.data.JSDOSession class on page 32 for a list of available events.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

The `subscribe( )` method throws an error object if *event-name* does not identify an event supported by the `JSDOSession` class (the look up is not case sensitive), or if an argument is not of the correct type.

See also:

unsubscribe( ) method (JSDOSession class) on page 51

unsubscribe( ) method (JSDOSession class)

Unsubscribes a given event handler function from an event of the current `JSDOSession` object.

This method throws an exception if the specified event is not supported by the `JSDOSession` class.

Return type: `undefined`

Applies to: progress.data.JSDOSession class on page 32

Syntax

```
unsubscribe ( event-name , event-handler [ , scope ] )
```

event-name

The name of a `JSDOSession` object event to which you unsubscribe an event handler. See the reference entry for the progress.data.JSDOSession class on page 32 for a list of available events.

event-handler

A reference to an event handler function that is to be removed from the list of callbacks that are called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function. Specifying the scope is optional in the event subscription. If the event subscription **does** specify an execution scope, you must specify a matching *scope* parameter when you call the `unsubscribe( )` method to cancel the event subscription.

The `unsubscribe( )` method throws an error object if *event-name* does not identify an event supported by the `JSDOSession` class (the look up is not case sensitive), if an argument is not of the correct type, or if there is no handler that was subscribed for that event with the same *scope* as passed to `unsubscribe( )`.

See also:

progress.data.JSDOSession class on page 32, subscribe( ) method (JSDOSession class) on page 50

unsubscribeAll( ) method

Unsubscribes all event handler functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event handler functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Return type: `null`

Applies to: `progress.data.JSDO` class, progress.data.JSDOSession class on page 32, `progress.data.Session` class

Syntax

```
unsubscribeAll ( [ event-name ] )
```

event-name

A `String` that if specified, is the name of an event on the current object from which to unsubscribe all event handlers. If not specified, the method unsubscribes all event handlers from all events of the current object. See the reference entry for the `progress.data.JSDO` class, progress.data.JSDOSession class on page 32 or the `progress.data.Session` class for a list of available events.

For a `JSDOSession` or `Session` object, the `unsubscribeAll( )` method throws an error object if *event-name* does not identify an event supported by the `progress.data.JSDOSession` or `progress.data.Session` class (the lookup is case insensitive), or if *event-name* is not a `String`. For a JSDO, the method ignores these conditions.

See also:

`progress.data.JSDO` class, `progress.data.Session` class, `unsubscribe( )` method (JSDO class), unsubscribe( ) method (JSDOSession class) on page 51, `unsubscribe( )` method (Session class)

JSDOSession properties

The following sections describe the properties of the `progress.data.JSDOSession` class.

authenticationModel property (JSDOSession class)

Returns a string constant that specifies the authentication model that the current `JSDOSession` object is using and was passed as an option to the object's class constructor.

Data type: `String`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32

Values that can be returned include:

- **`progress.data.Session.AUTH_TYPE_ANON`** — The Mobile Web application supports Anonymous access. No authentication is required. This is the default value if none is passed to the `JSDOSession` constructor.
- **`progress.data.Session.AUTH_TYPE_BASIC`** — The Mobile Web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the Web application's resources for you, you need to pass these credentials in a call to the `JSDOSession` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your Mobile App, either using a form of your own design or using a template provided by Progress Software Corp.
- **`progress.data.Session.AUTH_TYPE_FORM`** — The Mobile Web application uses Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the Web application itself sends a form to the client to get the credentials. However, when you have the `JSDOSession` object manage access to the Web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `JSDOSession` intercepts the form sent by the Web application and handles the authentication without that form being displayed.

If the Mobile Web application requires authentication, you must set this value correctly in the `JSDOSession` constructor to ensure that users can log in.

See also:

The constructor description for the progress.data.JSDOSession class on page 32, login( ) method (JSDOSession class) on page 41

catalogURIs property

Returns the list of URIs successfully used to load JSDO catalogs into the current `JSDOSession` or `Session` object.

Data type: `String` array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

This list includes the URI for each JSDO catalog loaded using the `addCatalog( )` method on the current `JSDOSession` or `Session` object. To return a corresponding list of Mobile services for which the JSDO catalogs are loaded, read the `services` property.

See also:

`addCatalog( )` method (Session class), addCatalog( ) method (JSDOSession class) on page 38, services property on page 59

clientContextId property

The value of the most recent client context identifier (CCID) that the current `JSDOSession` or `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.

If none has yet been found, the value is `null`.

Data type: `String`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

The `JSDOSession` or `Session` object automatically detects, stores, and returns the CCID sent by any appropriately configured Mobile Web application for which it has started a login session. This CCID is the same as the value of the `ClientContextId` property on the ABL `Progress.Lang.OERequestInfo` class-based object that is passed from an AppServer client (in this case, the Mobile Web application) to the AppServer that is executing a Mobile request.

Note: You can access this `OERequestInfo` object on the AppServer using the ABL `CURRENT-REQUEST-INFO` attribute of the AppServer `SESSION` system handle. This CCID value is also available as the ABL `SESSION-ID` attribute of the single sign-on (SSO) client-principal handle returned by the `GetClientPrincipal( )` method of the same ABL `OERequestInfo` class-based object.

Note: You can configure a Mobile Web application to send a CCID using the features for configuring any OpenEdge Web application to use SSO. For more information, see the sections on enabling SSO for a Web application in the administration documentation for the OpenEdge AppServer or Pacific Application Server for OpenEdge.

See also:

serviceURI property on page 60

connected property

Returns a `Boolean` that indicates the most recent online state of the current `JSDOSession` or `Session` object, when it last determined if the Mobile Web application it manages is available.

If the property value is `true`, the object most recently determined that the session is connected and logged in to its Mobile Web application. If its value is `false`, the session was last found to be disconnected. The default value is `false`.

Note:

Because of the dynamics of any network environment, the value of this property might not reflect the current status of the object's connection to its Mobile Web application. You can therefore invoke the object's `ping( )` method (either explicitly or automatically by setting the value of its `pingInterval` property) to update the object's most recent online status.

Data type: `Boolean`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

The most recent session online status determination might be identified from any of the following:

- A successful result of the `JSDOSession` or `Session` object executing its `login( )` method, which sets the property to `true`. Prior to calling `login( )`, the value of this property is `false`.
- A successful result of the `JSDOSession` or `Session` object executing its `logout( )` method, which sets the property to `false`.
- The `JSDOSession` or `Session` object receiving an `offline` or `online` event from its window object.
- A JSDO attempting to send a request to a Rollbase or OpenEdge Mobile service that the `JSDOSession` or `Session` object manages.
- The result of the `JSDOSession` or `Session` object executing its `ping( )` method.

See also:

login( ) method (JSDOSession class) on page 41, `login( )` method (Session class), offline event on page 61, online event on page 63, ping( ) method (JSDOSession class) on page 47, `ping( )` method (Session class)

JSDOs property

Returns an array of JSDOs that use the current `JSDOSession` or `Session` object to communicate with their mobile services.

Data type: JSDO array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

See also:

pingInterval property on page 58, services property on page 59

loginHttpStatus property

Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` or `Session` object.

Data type: `Number`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

See also:

login( ) method (JSDOSession class) on page 41, `login( )` method (Session class)

loginResult property

Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` or `Session` object.

Data type: `Number`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

Possible `loginResult` values include the following numeric constant values:

- **`progress.data.Session.LOGIN_SUCCESS`** — User login session started successfully.
- **`progress.data.Session.LOGIN_AUTHENTICATION_FAILURE`** — User login failed because of invalid user credentials.
- **`progress.data.Session.LOGIN_GENERAL_FAILURE`** — User login failed because of a non-authentication failure.

For a more specific status code returned in the HTTP response, you can check the value of the `loginHttpStatus` property.

The value of this property is `null` prior to the first login attempt and after a successful logout (until the next login attempt).

See also:

login( ) method (JSDOSession class) on page 41, `login( )` method (Session class), loginHttpStatus property on page 55

onOpenRequest property

Returns the reference to a user-defined callback function that the `JSDOSession` or `Session` object executes to modify a request object before sending the request object to the server.

For example, this function might add a message header by modifying the XMLHttpRequest object used to send the request.

You do not typically use this property, because OpenEdge Mobile properly handles preparation of the request object for normal circumstances.

Data type: `function`

Access: Readable/Writable

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

By default, the value of the `onOpenRequest` property is `null`, meaning that the request object is sent without modification. If the value is set to a callback function, the function takes a single object parameter.

Syntax

This is the syntax for setting this property to a function reference:

```
mySession.onOpenRequest = funcRef
```

mySession

A reference to the `JSDOSession` or `Session` object for which the request object is to be modified before sending a request to the server.

funcRef

A reference to a JavaScript callback function that has the following signature:

Syntax:

```
function [ func-name ] ( param )
```

func-name

The optional name of a function you have defined external to the property assignment. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

param

An `Object` that has the following properties:

- **xhr** — An object reference to the XMLHttpRequest object (XHR) used to send the request. The current request object can be modified by the function. When the callback is called, `XMLHttpRequest.open( )` will already have been called on the XHR , but the callback can call `open( )` again, overriding the effects of the first `open( )` call. When the callback function is used for a `login( )`, `addCatalog( )`, or `logout( )` call, although it should not be necessary and is not recommended, it is possible to replace the XHR entirely by creating a new object and assigning it as the value of the `xhr` property.
- **verb** — The HTTP operation (GET, PUT, etc.) to be performed by the request.
- **uri** — The URI to which the request is addressed.
- **session** — A reference to the `Session` object that invoked the callback.
- **formPreTest** — A `Boolean` specifying whether the current `login( )` request is a preliminary request, used in cases of Form authentication, to determine whether the user is already logged in (`true`) or an actual login request (`false`).
- **async** — A `Boolean` specifying whether the request is asynchronous (`true`) or synchronous (`false`).

Caution: For a `JSDOSession` object, if the callback function is used for a `login( )`, `addCatalog( )`, or `logout( )` call, and if it calls `XMLHttpRequest.open( )`, the `async` property value passed to that `open( )` call must be `true` (i.e., the `open( )` method must specify that the request is to be sent asynchronously).

Caution: For a `Session` object, if the callback function is used for a `login( )`, `addCatalog( )`, or `logout( )` call, and if it calls `XMLHttpRequest.open( )`, the `async` property value passed to that `open( )` call must match the `async` value that was passed to the `login( )`, `addCatalog( )`, or `logout( )` call.

If you assign a callback function as the value of `onOpenRequest`, it remains in effect for all requests for the duration of the session unless it is replaced by another function or is set to `null`.

Example

Be sure to reset the value of the property as necessary, as in the following example:

```
myJSDOsession.onOpenRequest = function( params ) {
    params.xhr.setRequestHeader('Authorization', auth);
};

myJSDOsession.login(usuernamе, password).done(
    function(JSDOsession, result, info) {
        JSDOsession.onOpenRequest = null;
        . . .
}).fail(
    function(JSDOsession, result, info) {
        alert("Login failed");
});
```

See also:

request object, `xhr` property

pingInterval property

A `Number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` or `Session` object's `ping( )` method and the next.

Setting this property to a value greater than zero (0) causes the `JSDOSession` or `Session` object to begin executing its `ping( )` method, and when execution completes, to repeatedly execute the method after the specified delay. If you set its value to zero (0), no further execution of `ping( )` occurs after any current execution completes. The default value is zero (0).

Data type: `Number`

Access: Readable/Writable

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

You can set `pingInterval` to start the automatic execution of `ping( )` any time after you create the `JSDOSession` or `Session` object. However, `ping( )` does not begin executing until and unless you have successfully invoked the object's `login( )` method to start a user login session.

Note that when you call the `ping( )` method directly, you have several options for how to call it to get the results. You do not have these options and you cannot get results directly from each automatic execution of `ping( )` that begins from a setting of `pingInterval`. The effects from this automatic execution are limited to causing the `JSDOSession` or `Session` object to fire its `offline` or `online` event, and to change the value of its `connected` property, when a given `ping( )` execution detects a change in the object's online status.

See also:

connected property on page 54, JSDOs property on page 55, login( ) method (JSDOSession class) on page 41, `login( )` method (Session class), offline event on page 61, online event on page 63, ping( ) method (JSDOSession class) on page 47, `ping( )` method (Session class)

services property

Returns an array of objects that identifies the Mobile services that have been loaded for the current `JSDOSession` or `Session` object and its Mobile Web application.

Data type: `Object` array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

You load Mobile services for a `JSDOSession` or `Session` object by loading the corresponding JSDO catalogs using the object's `addCatalog( )` method. For a `JSDOSession` object, JSDO catalogs can be loaded either before or after you log into a Mobile Web application using the object's `login( )` method. For a `Session` object, JSDO catalogs must be loaded after you log into a Mobile Web application using the object's `login( )` method.

Each object in the array returned by this session property contains two properties:

- **`name`** — The name of a Mobile service
- **`uri`** — The URI for the service. If the address of the service in the catalog is an absolute URI, this value is that URI. If the service address is relative, this value is the relative address concatenated to the value of the `JSDOSession` or `Session` object's `serviceURI` property, which contains the Mobile Web application URI used by the object's `login( )` method.

Note: To return a corresponding list of URIs for the loaded JSDO catalogs, read the `catalogURIs` property.

Example

Given the following service names and URIs loaded into a `JSDOSession` object from unsecured locations:

- **`"CustomerSvc"`** service with this URI: **`"/rest/CustomerSvc"`**
- **`"ItemSvc"`** service with this URI: **`"http://itemhost:8080/SportsApp/rest/ItemSvc"`**

The following code fragment produces the output that follows:

```
// create Session
pdsession = new progress.data.JSDOSession('http://custhost:8080/SportsApp');

window.loginView = kendo.observable({
    submit: function() {

        // load 2 unprotected catalogs
        pdsession.addCatalog([ "/SportsApp/static/mobile/CustomerSvc.json",
              "http://itemhost:8080/SportsApp/static/mobile/ItemSvc.json" ]
            ).done(
            function(JSDOsession, result, info) {

                // log in anonymously
                JSDOSession.login(
                    ).done(
                    function(JSDOSession, result, info) {

                        /* Use services property to print services
                           loaded by this Session object */
                        for (var i=0; i < JSDOSession.services.length; i++)
{
                            console.log( JSDOSession.services[i].name + "
"
                                         + JSDOSession.services[i].uri );
                        }
                        // Stuff with widgets and DataSources . . .
                    }).fail( // log in
                    function(JSDOSession, result, info) { . . .
                    });
            }).fail( // Add catalogs
            function(JSDOSession, result, details) { . . .
            });
    }
});
```

Output from the preceding code fragment:

```
CustomerSvc   http://custhost:8080/SportsApp/rest/CustomerSvc
ItemSvc   http://itemhost:8080/SportsApp/rest/ItemSvc
```

See also:

addCatalog( ) method (JSDOSession class) on page 38, `addCatalog( )` method (Session class), catalogURIs property on page 53, login( ) method (JSDOSession class) on page 41, `login( )` method (Session class), serviceURI property on page 60

serviceURI property

Returns the URI to the Mobile Web application that has been passed as an option to the class constructor for the current `JSDOSession` object or that has been passed as a parameter to the most recent call to the `login( )` method on the current `Session` object, whether or not the most recent call to `login( )` succeeded.

Data type: `String`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

See also:

login( ) method (Session class)

userName property

Returns the user ID passed as a parameter to the most recent call to the `login( )` method on the current `JSDOSession` or `Session` object.

Data type: `String`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

This value is returned, whether or not the most recent call to `login( )` succeeded.

Note: In atypical cases, this property might not specify the name of the user logged in for the current session. The logged-in user can be different from this property setting if the authentication was done by the browser or hybrid native wrapper prior to the `login( )` method being called, or if the `login( )` method was passed incorrect user credentials and the browser or native wrapper took over and completed the user authentication.

See also:

login( ) method (JSDOSession class) on page 41, `login( )` method (Session class)

JSDOSession events

The following sections describe the events of the `progress.data.JSDOSession` class.

offline event

Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone offline, or that the Mobile Web application to which it has been connected is no longer available.

This event always fires when the device on which the `JSDOSession` or `Session` object is created goes offline (that is, the device is disconnected from the network). For this event to fire when the Mobile Web application to which it has been connected is no longer available, the `JSDOSession` or `Session` object must have previously:

1. Been connected to the Web application using the object's `login( )` method
2. Not been disconnected from the Web application using the object's `logout( )` method

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

The following parameters appear in the signature of the event handler function:

Syntax

```
function ( session , off-line-reason , request )
```

session

A reference to the `JSDOSession` or `Session` object that has detected the offline condition.

off-line-reason

A string constant indicating the reason that the offline event has been fired. Possible constants include:

- **`progress.data.Session.APPSERVER_OFFLINE`** — (OpenEdge Mobile services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
- **`progress.data.Session.DEVICE_OFFLINE`** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.

  Note: This condition will fire the `offline` event on a `JSDOSession` or `Session` object even if the object has not yet invoked its `login( )` method.

- **`progress.data.Session.SERVER_OFFLINE`** — The Web server is not available. For a Rollbase Mobile service, this is the Web server for the public or private cloud. For an OpenEdge Mobile service, this is the Tomcat Java servlet container.
- **`progress.data.Session.WEB_APPLICATION_OFFLINE`** — The server is running, but the Java Web application that implements the Mobile service is not deployed.

You can use the string values of these constants directly to display or log messages, or test the values only and respond to the result in some other way.

request

If the offline condition was detected as a result of a request sent on behalf of a JSDO, this is a reference to the `request` object used to make the request. For more information, see the description of the `request` object in *OpenEdge Development: Mobile Applications*. If the offline event is the result of the device going offline or because of a call to the `ping( )` method (either directly or because the `pingInterval` property is set greater than 0), this parameter is `null`.

Note: If the underlying database accessed by the Rollbase cloud or the OpenEdge AppServer is unavailable, this will **not** cause the `offline` event to fire, nor will the fact that a particular Mobile service contained by the hosting Web application was not loaded when the application started.

Example

The following code fragment subscribes the function, `onSessionOffline`, to handle the `offline` event fired on the session, `mySession`:

```
/* subscribe to the offline event */
var mySession = new progress.data.Session;
mySession.subscribe('offline', onSessionOffline );

/* some code that might cause mySession to detect that the
   application is offline */

. . .

function onSessionOffline( session , offlineReason , request ) {

  switch( offlineReason ) {
  case progress.data.Session.APPSERVER_OFFLINE:
    document.write(progress.data.Session.APPSERVER_OFFLINE);
    . . .
    break;
  case progress.data.Session.DEVICE_OFFLINE:
    document.write(progress.data.Session.DEVICE_OFFLINE);
    . . .
    break;
  case progress.data.Session.SERVER_OFFLINE:
    document.write(progress.data.Session.SERVER_OFFLINE);
    . . .
    break;
  case progress.data.Session.WEB_APPLICATION_OFFLINE:
    document.write(progress.data.Session.WEB_APPLICATION_OFFLINE);
    . . .
    break;
  default:
    document.write("ERROR: An offline event returned an unknown reason: " +
offLineReason);
  };

};
```

See also:

connected property on page 54, login( ) method (JSDOSession class) on page 41, `login( )` method (Session class), logout( ) method (JSDOSession class) on page 45, `logout( )` method (Session class), online event on page 63, ping( ) method (JSDOSession class) on page 47, `ping( )` method (Session class), subscribe( ) method (JSDOSession class) on page 50, `subscribe( )` method (Session class), unsubscribe( ) method (JSDOSession class) on page 51, `unsubscribe( )` method (Session class)

online event

Fires when the current `JSDOSession` or `Session` object detects that the device on which it is running has gone online after it was previously offline, or that the Mobile Web application to which it is connected is now available after it was previously unavailable.

This event always fires when the device on which the `Session` object is created goes online after having been offline (that is, the device is reconnected to the network). For this event to fire when the Mobile Web application to which it has been connected is now available after having been unavailable, the `Session` object must have previously:

1. Been connected to the Web application using the object's `login( )` method
2. Not been disconnected from the Web application using the object's `logout( )` method
3. Detected an `offline` event as a result of sending a Mobile service request to the Web application or executing the `ping( )` method

Applies to: progress.data.JSDOSession class on page 32, `progress.data.Session` class

The following parameters appear in the signature of the event handler function:

Syntax

```
function ( session , request )
```

session

A reference to the `JSDOSession` or `Session` object that has detected the online condition.

request

If the online condition was detected as a result of a request sent on behalf of a JSDO, this is a reference to the `request` object used to make the request. For more information, see the description of the `request` object in *OpenEdge Development: Mobile Applications*. If the online event is the result of the device itself coming back online or because of a call to the `ping( )` method (either directly or because the `pingInterval` property is set greater than 0), this parameter is `null`.

This event can fire because:

- The device on which the App is running was previously offline (in airplane mode or could not connect to a wi-fi network), but is now online again.

  Note: This condition will fire the `online` event on a `JSDOSession` or `Session` object even if the object has not yet invoked its `login( )` method or has already invoked its `logout( )` method .

- A `JSDOSession` or `Session` object that previously fired its `offline` event because of a failed attempt to contact a Mobile service has now been used successfully to contact that same Mobile service.

Example

The following code fragment subscribes the function, `onSessionOnline`, to handle the `online` event fired on the session, `mySession`:

```
/* subscribe to the online event */
var mySession = new progress.data.Session;
mySession.subscribe('online', onSessionOnline );

/* some code that might cause mySession to detect that the
   application is online */

. . .

function onSessionOnline( session , request ) {

  document.write("Your best Mobile App is back to work! "
                 + "You might be prompted to login again.");

};
```

See also:

connected property on page 54, login( ) method (JSDOSession class) on page 41, `login( )` method (Session class), logout( ) method (JSDOSession class) on page 45, `logout( )` method (Session class), offline event on page 61, ping( ) method (JSDOSession class) on page 47, `ping( )` method (Session class), subscribe( ) method (JSDOSession class) on page 50, `subscribe( )` method (Session class), unsubscribe( ) method (JSDOSession class) on page 51, `unsubscribe( )` method (Session class)

Note about certificate management for HTTPS testing

The SSL certificate provided with OpenEdge for HTTPS testing is a self-signed certificate. Mobile client platforms are becoming increasingly strict regrading self-signed certificates. Some will prompt the end user with a warning when they receive a self-signed certificate, others might simply fail a login attempt. You should (and will probably have to) do your HTTPS testing with a certificate signed by a trusted Certificate Authority (CA).

Behavior changes in the OpenEdge JSDO

Since OpenEdge 11.5 was released, a feature of the OpenEdge JSDO has changed its behavior from what is documented in the OpenEdge Release 11.5 documentation. As currently documented, when a JSDO object is created for a ProDataSet resource that supports before-imaging, any Mobile create, update, delete, or submit operation can have a successful result with errors returned in the changed records. That is, it is documented that after invoking the JSDO `saveChanges( )` method, the `success` property of the request object returned from the server for each Mobile operation on before-image data can be `true` and the associated change record can also contain an error string. You can then return this error string using the `_errorString` property accessed from the data of the change `JSRecord` object or using the `getErrorString( )` method invoked on the change `JSRecord` object.

The undocumented change in behavior since OpenEdge 11.5 is that this error string can only be found in a change record when the `success` property returned in the request object is `false`, like any other error returned from a Mobile operation. With this change, the `success` property behaves consistently, returning `true` only when **no** errors are returned for the Mobile operation, and returning `false` when **any** errors, from **any** source, are returned for the Mobile operation.