Cloud Data Object API Specification

Progress is committed to delivering market-leading technology innovations that empower our partners and customers to dramatically improve the development, deployment, integration and management of their business applications. The Cloud Data Object brings data management features normally found in Systems of Records to client applications in the cloud, specifically web and mobile apps.

Samples in this specification use JavaScript on the client where the Cloud Data Object is referred to as a JavaScript Data Object (JSDO) and OpenEdge ABL on the server. In reality, any client or server language can be used to implement this specification. It is expected that when implementing specifically for your application, that you will be using your own namespace instead of CDO or Progress' JSDO.

Document History:

VERSION	DATE	CHANGE DESCRIPTION
1.0	03/31/2015	Initial version
1.1	08/01/2015	Update syntax for Release 4.1 additions
1.2	02/12/2016	Update syntax for Release 4.2 additions

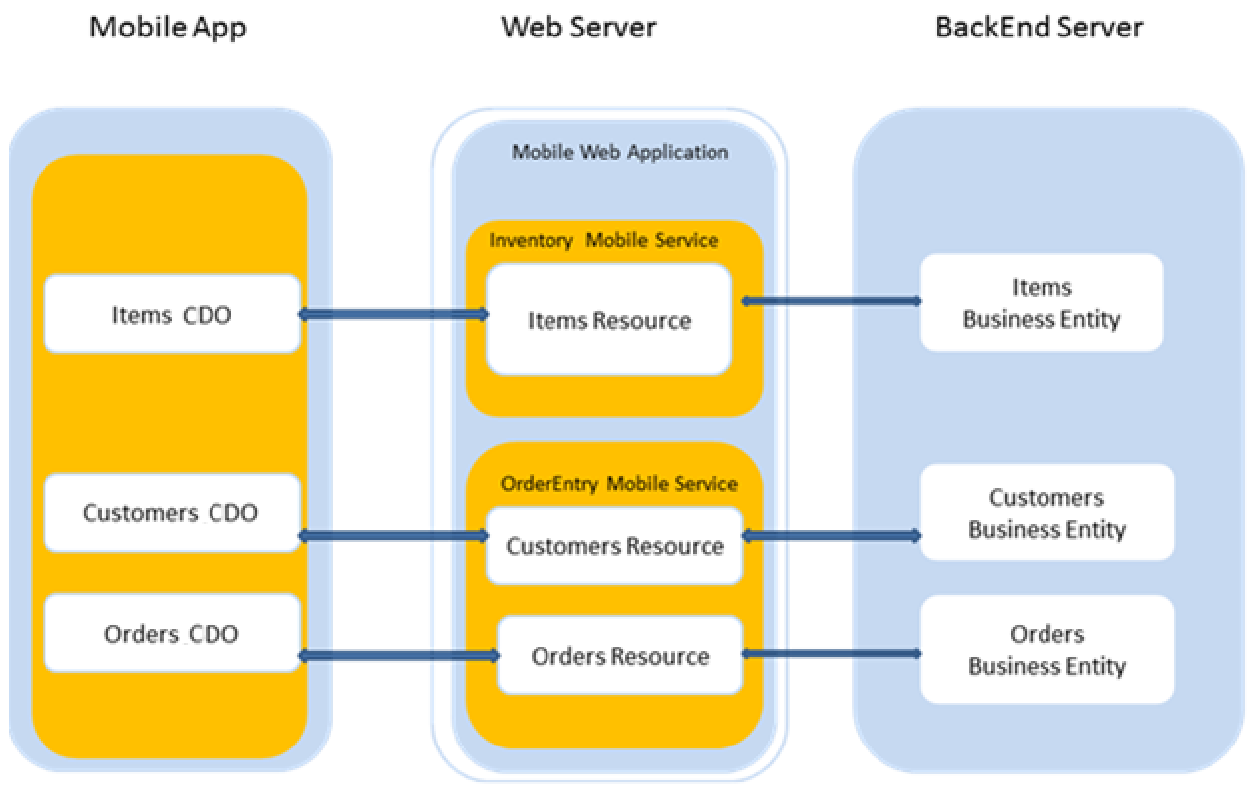
Contents

Introduction

Web and mobile applications often communicate with a server for data and business logic. This specification introduces the *Cloud Data Object* (CDO), a client-side object that manages transactional data updates and access to server-side business logic.

A *Cloud Data Service* (CDS) defines the API for one or more server-side resources. Each resource provides access to a logical schema and its related business logic, which for the purposes of this document will be referred to as a *Business Entity*. The supported schemas are a single table or a dataset which contains one or more tables, where the tables may be related. A Business Entity is implemented using a standard set of built-in operations to read and modify data on the backend. A Business Entity can also provide additional (customized) methods, so the resource will also provide access to these non-standard methods.

A CDO provides client access to the data and operations of a single resource. Client code can call methods on a CDO to execute the mapped server-side operations on the backend. These server-side operations are also refered to as Data Object operations. The data for these operations is serialized between the Client and the Server.
For information on the *CDO Catalog file,* please refer to the document Cloud Data Object Catalog Specification.



The figure above provides an overview of how a CDO accesses a resource which works the same for a given CDO regardless of the type of client or type of backend.

Classes to support the Cloud Data Object

The following is a detailed explanation of the classes that support the Cloud Data Object. The classes are:

- CloudDataRecord
- CloudDataObject
- CloudSession
- request

CloudDataRecord Class

`CloudDataRecord` is a record instance for any table stored in the local memory of an associated class instance (CDO).

Properties

Member	Brief description (See also the reference entry)
_errorString property	A string value available on the `data` property of every `CloudDataRecord object` in CDO memory that is set as part of before-image data for the record and provides descriptive information about any record change error on the server following a Data Object create, update, delete, or submit operation. NOTE: Usage of _errorString property has been deprecated. Use getErrorString() method instead.
_id property	A string value available on the `data` property of every `CloudDataRecord object` in CDO memory that provides a unique internal ID for the record. NOTE: Usage of _id property has been deprecated. Use getId() method instead.
data property	The data (field values) for a record associated with a `CloudDataRecord object`.

Methods

Member	Brief description (See also the reference entry)
acceptRowChanges( ) method	Accepts changes to the data in CDO memory for a specified record object.
assign( ) method (CDO class) alias: update	Updates field values for the specified `CloudDataRecord` object in CDO memory.
getErrorString( ) method	Returns any before-image error string in the data of a record object referenced in CDO memory that was set as the result of a Data Object create, update, delete, or submit operation.
getId( ) method	Returns the unique internal ID for the record object referenced in CDO memory.
remove( ) method	Deletes the specified table record referenced in CDO memory.
rejectRowChanges( ) method	Rejects changes to the data in CDO memory for a specified record object.

Events

This object has no events.

Example

The following example assumes that a CDO is referenced by the `cdo` variable. The example creates a new record object along with a message with credit information using properties of the record object:

```
function addRecord() {
  var record = cdo.add( {Balance: 10000, State: 'MA'} );
  alert( 'Record ID: ' + record.getId( ) + ' CreditLimit: ' +
         record.data.CreditLimit );
}
```

Note: Using the `add( )`, `find( )`, `findById( )`, or `foreach( )` method, or the `record` property, on a given CDO and table reference, a `CloudDataRecord` instance returns a working record for the table referenced in CDO memory. You can then use properties and methods of the `CloudDataRecord` to update, delete, or display the specified record from the CDO.

CloudDataObject Class

The `CDO` provides access to resources in a Cloud Data Service, known as a *Cloud Data Resource*. A single `CDO` object (CDO) provides access to one resource in a Cloud Data Service. The CDO provides application-oriented methods on the client to work with data and invoke business logic methods on a Cloud Data Service. The supported *Cloud Data operation* types are:

1. CRUD(Create, Read, Update, Delete)
2. Submit
3. Invoke operations

You identify how the CDO maps methods to operations of a given resource in a *CDO catalog file* that identifies how a CDO that you create can access the corresponding resource using methods of the CDO.

At run time, the CDO maintains an internal data store (*CDO memory*) for managing table data that is exchanged between the server and client, and it provides methods to read and write the data in CDO memory as individual record objects. To support this data exchange, a resource can be organized into basic operation types that include *built-in* create, read, update, delete (CRUD), and submit operations, and *non-built-in* invoke operations. The built-in Data Object operations can operate on a single table or on a single DataSet containing one or more tables. Each built-in operation type maps to a corresponding built-in method of the CDO.

The records of each table are presented as an array of record objects, which the built-in methods use to exchange the data with the Cloud Data Server. The built-in methods, through their corresponding operation types, support the common business operations that can be generated directly from a Business Entity. Other methods of the CDO provide access to individual record objects of CDO memory. Based on the results of its methods, the CDO also maintains a working record for each table in its memory store that you can access directly using table and field references (see the notes). Thus, using the methods of a CDO and its table references, you can interact with a corresponding resource in a consistent manner from one resource (and its corresponding CDO) to another.

A CDO also supports non-built-in invoke operations that allow specific routines to be exposed in a resource and executed as corresponding JavaScript methods. You can do this in Developer Studio by annotating routines specifically as invoke operations. You can then call each routine annotated as an invoke operation using a unique *invocation method* on the CDO. Note that data exchanged between the Cloud Data Server and client using invoke operations is not automatically stored in CDO memory. It is initially accessible only through parameters and return values of the invocation methods provided by the CDO. You can subsequently use CDO methods to exchange data between the invocation methods and CDO memory, which is maintained and synchronized with the Cloud Data Server using the CDO built-in methods.

When you instantiate a CDO, it relies on a prior login session that you can establish using an instance of the Cloud`Session` class. This login session enables optionally secure communications between the client CDO and the Web server, specified Cloud Data Services, and ultimately the Cloud Data Server that implements the resource accessed by the CDO.

Constructors

Two constructors are available for the CDO. The first constructor takes the name of the corresponding resource as a parameter; the second constructor takes an initialization object as a parameter.

The resource name specified for the constructor must match the name of a resource provided by a Cloud Data Service for which a login session has already been started. After the CDO is created, it uses the information stored in the CDO catalog that is loaded for the Cloud Data Service to communicate with the specified resource.

Syntax

```
CDO ( resource-name )
CDO ( init-object )
```

`resource-name`

A string expression set to the name of a resource provided by a Cloud Data Service for which a login session has been started.

`init-object`

An object that can contain any writable CDO properties. It **must** contain the required CDO `name` property, which specifies the resource for the CDO. It can also contain either or both of the following initialization properties:

- **`autoFill`** — A `Boolean` that specifies whether the CDO invokes its `fill( )` method upon instantiation to initialize CDO memory with data from the resource. The default value is `false`.
- **`events`** — An object that specifies one or more CDO event subscriptions, each with its properties represented as an array, with the following syntax:

Syntax

```
events : {
     'event' : [ {
          [ scope : object-ref , ]
          fn : function-ref
     } ] [ ,
     'event' : [ {
          [ scope : object-ref , ]
          fn : function-ref
     } ] ] . . .
}
```

`event`

The name of an event the CDO instance subscribes to. See Events for a list of available CDO events.

object-ref

An optional object reference that defines the execution scope of the 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).

function-ref

A reference to an event handler function, that is called when the event fires. Each event passes a fixed set of parameters to its event handler, as described in the reference entry for the event in CDO Properties, Methods, and Events Reference.

Example

The following example illustrates the use of an initialization object to instantiate a CDO:

```
dsItems = new progress.data.CDO({
  name : 'Item',
  autoFill : false,
  events : {
    'afterFill' : [ {
                       scope : this,
                       fn : function (cdo, success, request) {
                         // afterFill event handler statements ...
                       }
                    } ]
  }
});
```

Properties

Member	Brief description (See also the reference entry)
autoApplyChanges property	A `Boolean` on a CDO that indicates if the CDO automatically accepts or rejects changes to CDO memory when you call the `saveChanges( )` method.
autoSort property	A `Boolean` on a CDO and its table references that indicates if record objects are sorted automatically on the affected table references in CDO memory at the completion of a supported CDO operation.

Member	Brief description (See also the reference entry)
caseSensitive property	A `Boolean` on a CDO and its table references that indicates if `String` field comparisons performed by supported CDO operations are case sensitive or case-insensitive for the affected table references in CDO memory.
name property	The name of the resource for which the current CDO is created.
record property	A property on a CDO table reference that references a `CloudDataRecord object` with the data for the working record of a table referenced in CDO memory.
table reference property	An object reference property on a CDO that has the name of a table mapped by the resource to a table for which the current CDO is created.
useRelationships property	A `Boolean` that specifies whether CDO methods that operate on table references in CDO memory work with the table relationships defined in the schema (that is, work only on the records of a child table that are related to the parent).

Methods

Certain methods of the `CDO` class are called on the CDO object itself, without regard to a table reference, whether that reference is explicit (specified in the method signature) or implicit (in the case of a CDO containing only one table that is not explicitly specified). Other methods can be called on a reference to a table mapped by the resource for which the current CDO is created.

Member	Brief description (See also the reference entry)
acceptChanges( ) method	Accepts changes to the data in CDO memory for the specified table reference or for all table references of the specified CDO.
acceptRowChanges( ) method	Accepts changes to the data in CDO memory for a specified record object.
add( ) method alias: create	Creates a new record object for a table referenced in CDO memory and returns a reference to the new record.

Member	Brief description (See also the reference entry)
addLocalRecords( ) method	Reads the record objects stored in the specified local storage area and updates CDO memory based on these record objects, including any pending changes and before-image data, if they exist.
addRecords( ) method	Reads an array, table, or ProDataSet object containing one or more record objects and updates CDO memory based on these record objects, including any pending changes and before-image data, if they exist.
assign( ) method alias: update	Updates field values for the specified `CloudDataRecord object` in CDO memory.
create( ) method	Creates a new record object for a table referenced in CDO memory and returns a reference to the new record.
deleteLocal( ) method	Clears out all data and changes stored in a specified local storage area, and removes the cleared storage area.
fill( ) method alias: read	Initializes CDO memory with record objects from the data records in a single table, or in one or more tables of a ProDataSet, as returned by the built-in read operation of the resource for which the CDO is created.
find( ) method	Searches for a record in a table referenced in CDO memory and returns a reference to that record if found. If no record is found, it returns `null`.
findById( ) method	Locates and returns the record in CDO memory with the internal ID you specify.
foreach( ) method	Loops through the records of a table referenced in CDO memory and invokes a user-defined callback function as a parameter on each iteration.
getData( ) method	Returns an array of record objects for a table referenced in CDO memory.

Member	Brief description (See also the reference entry)
getErrors() method	Returns an array of errors from the most recent CDO operation.
getErrorString( ) method	Returns any before-image error string in the data of a record object referenced in CDO memory that was set as the result of a Data Object create, update, delete, or submit operation.
getId( ) method	Returns the unique internal ID for the record object referenced in CDO memory.
getSchema( ) method	Returns an array of objects, one for each field that defines the schema of a table referenced in CDO memory.
hasData( ) method	Returns `true` if record objects can be found in any of the tables referenced in CDO memory (with or without pending changes), or in only the single table referenced on the CDO, depending on how the method is called; and returns `false` if no record objects are found in either case.
hasChanges( ) method	Returns `true` if CDO memory contains any pending changes (with or without before-image data), and returns `false` if CDO memory has no pending changes.
invocation method	Any method on the CDO that is defined by the resource to execute a corresponding routine on the Cloud Data
invoke() method	Asynchronously calls a custom invocation method on the CDO to execute an Invoke operation defined by a Data Object resource.
read( ) method	Initializes CDO memory with record objects from the data records in a single table, or in one or more tables of a ProDataSet, as returned by the built-in read operation of the resource for which the CDO is created.

Member	Brief description (See also the reference entry)
readLocal( ) method	Clears out the data in CDO memory and replaces it with all the data stored in a specified local storage area, including any pending changes and before-image data, if they exist.
rejectChanges( ) method	Rejects changes to the data in CDO memory for the specified table reference or for all table references of the specified CDO.
rejectRowChanges( ) method	Rejects changes to the data in CDO memory for a specified record object.
remove( ) method	Deletes the specified table record referenced in CDO memory.
saveChanges( ) method	Synchronizes to the Cloud Data Server all changes pending in CDO memory since the last call to the `fill( )` or `saveChanges( )` methods, or since any prior changes have been otherwise accepted or rejected.
saveLocal( ) method	Saves CDO memory to a specified local storage area, including pending changes and any before-image data, according to a specified data mode.
setSortFields( ) method	Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).
setSortFn( ) method	Specifies or clears a user-defined sort function with which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).
sort( ) method	Sorts the existing record objects for a table reference in CDO memory using either specified sort fields or a specified user-defined sort function.

Member	Brief description (See also the reference entry)
subscribe( ) method (CDO class)	Subscribes a given event handler function to a named event of the current CDO or table reference.
unsubscribe( ) method (CDO class)	Unsubscribes a given event handler function from a named event of the current CDO or table reference.

Events

Member	Brief description (See also the reference entry)
afterCreate event	Fires after the CDO, by means of a `saveChanges( )` call following an `add() or create()` call, sends a request to create a record and receives a response to this request from the Cloud Data Server.
afterDelete event	Fires after the CDO, by means of a `saveChanges( )` call following a remove() call, sends a request to delete a record and receives a response to this request from the Cloud Data Server.
afterFill event	Fires after the CDO, by means of a `fill( )` call, sends a request to read a table or ProDataSet into CDO memory and receives a response to this request from the Cloud Data Server.
afterInvoke event	Fires after a non-built-in method is called asynchronously on a CDO and a response to the request is received from the Cloud Data Server.
afterRead event	Fires after the CDO, by means of a `read( )` call, sends a request to read a table or ProDataSet into CDO memory and receives a response to this request from the Cloud Data Server.
afterSaveChanges event	Fires once for each call to the `saveChanges( )` method on a CDO, after responses to all create, update, and delete requests have been received from the Cloud Data Server.
afterUpdate event	Fires after the CDO, by means of a `saveChanges( )` call following an `assign( )` or update() call, sends a request to update a record and receives a response to this request from the Cloud Data Server.
beforeCreate event	Fires before the CDO, by means of a `saveChanges( )` call making an `add( )` or create() call, sends a request the Cloud Data Server to create a record.
beforeDelete event	Fires before the CDO, by means of a `saveChanges( )` call making a `remove( )` call, sends a request the Cloud Data Server to delete a record.
beforeFill event	Fires before the CDO, by means of a `fill( )` call, sends a request to the Cloud Data Server to read a table or ProDataSet into CDO memory.

Member	Brief description (See also the reference entry)
beforeInvoke event	Fires when a non-built-in method is called asynchronously on a CDO, before the request for the operation is sent to the Cloud Data Server.
beforeRead event	Fires before the CDO, by means of a `read( )` call, sends a request to the Cloud Data Server to read a table or ProDataSet into CDO memory.
beforeSaveChanges event	Fires once for each call to the `saveChanges( )` method on a CDO, before any create, update, or delete requests are sent to the Cloud Data Server.
beforeUpdate event	Fires before the CDO, by means of a `saveChanges( )` call making an `assign( )` or update() call, sends a request the Cloud Data Server to update a record.

The CDO can subscribe to the events listed in the previous table in either of two ways:

- Subscription via CDO constructor — In the *`init-object`* parameter of the constructor, list each subscribed event with an optional scope object and an event handler method to be executed when the event fires. See the constructors' description for this class.
- Subscription via subscribe( ) method — See subscribe( ) method (CDO class).

Note: CDO events do not fire if the method call is synchronous.

Example

The following example reads customer records from a server-side table or ProDataSet and displays fields from the records in a list on the current Web page:

```
var session = new progress.data.Session();

// assuming userName and password came from the UI
var loginResult = session.login('/MobileApp', userName, password );
if (loginResult != progress.data.Session.LOGIN_SUCCESS) {
    //process login failure here and return/throw error, etc
    throw new Error("Login failed");
}
session.addCatalog('/MobileApp/static/mobile/MobileSvc.json');
var cdoOrderEntry = new progress.data.CDO( 'OrderEntry' );
cdoOrderEntry.subscribe('AfterFill', onAfterFill);

function onAfterFill() {
    });
}
```

Notes

- The CDO supports a working record for each table referenced in CDO memory. Certain methods set a specific record as the working record. After other methods execute, there is no working record or existing working records remain unchanged. When there is a working record, you can access the fields of the record using one of the following mechanisms:

Syntax

```
cdo-ref.table-ref.field-ref
cdo-ref.record.data.field-ref // Read-only;
                               // For a cdo-ref with only one table-ref
cdo-ref.table-ref.record.data.field-ref // Read-only
record-ref.data.field-ref // Read-only
```

cdo-ref

The reference to a CDO, and if the CDO references only one table, an implied reference to the working record defined for that table.

`table-ref`

A table reference with the name of a table in *`cdo-ref`* memory and a reference to the table working record. There is one table reference in a CDO for each table referenced by the CDO.

`field-ref`

A field reference on a *`table-ref`*, or a property on the `data` property object, with the name and value of a field in the working record of the referenced table. There is one such field reference and `data` object property for each field defined in the table schema.

`record`

A property of type `CloudDataRecord` on a table reference, which references the working record of a referenced table specified by:

- *`cdo-ref.table-ref`*
- *`cdo-ref`* if the CDO references only one table

If the CDO references more than one table, the `record` property is `null` at the CDO level and is available only on a *`table-ref`*.

`data`

A property on a `CloudDataRecord object` with the field values for the working record specified by:

- *`cdo-ref.table-ref`*
- *`cdo-ref`* if the CDO references only one table
- A *`record-ref`* returned for an associated CDO table reference

Note: If a *`field-ref`* has the same name as a built-in property or method of the CDO, you **must** use the `data` property to reference its value in the working record.

Caution: Never write directly to a *`field-ref`* using this `data` property; in this case, use *`field-ref`* **only to read** the data. Writing field values using the `data` property does **not** mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in CDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a *`cdo-ref.table-ref.field-ref`* for a working record or by calling the `assign( )` method on a valid *`table-ref`* or `CloudDataRecord object` reference.

`record-ref`

A reference to a `CloudDataRecord` object for a table referenced in CDO memory. You can return a *`record-ref`* for a working record as the value of the `record` property or as a value returned by supported CDO built-in methods that return a working record, such as `add( )` and `find( )`.

For more information on properties available to reference working record fields using this syntax, see the properties listed in this reference entry and in CloudDataRecord object reference entry. For more information on the methods for setting the working record for referenced tables, see methods listed in this reference entry and CloudDataRecord object reference entry.

- Many CDO built-in methods are actually invoked on a CDO table reference, and can only be invoked on the CDO itself when its CDO memory is initialized with a single table.

- For a multi-table ProDataSet, the CDO accesses the data for all unrelated tables in CDO memory as top-level tables of the CDO. Access to data for all related child tables depends on the working record of the parent table in the CDO and the setting of the `useRelationships` property.

See also

CloudDataRecord object, Session class, record property, table reference property (CDO class)

CDOSession Class

The `CDOSession` class can manage user authentication and session identification information in HTTP/S messages sent between `CDO` objects (CDOs) running in an App and Cloud Data Services running on a Web server. The authentication information includes a user ID and password (*user credentials*). The session identification information includes a URI, which identifies the Mobile or Web application that provides the REST transport between its defined set of Cloud Data Services and the client that accesses them, and possibly a session ID which identifies the user login session for the entire set of Cloud Data Services supported by the Mobile or Web application.

To start a user login session, invoke the `login( )` method on a `CDOSession` object that you have instantiated, passing user credentials if necessary and an optional options parameter. Once started, a login session for a Mobileor Web application supports all Cloud Data Services that the application provides, each of which can provide one or more resources.

Each Cloud Data Service provided by a Mobile or Web application relies on a separate CDO catalog file to define the communications between its resources and the CDOs that access them from the client. Once a user login session is established for the application, you can use its `CDOSession` object to load the catalog for each Cloud Data Service provided by the Web application. Once the CDO catalog is loaded for the Cloud Data Service, you can instantiate a CDO to access any resource provided by the service in the catalog. If required, the authentication information for the session is also used to authorize access to the resource by its CDO.

All CDOs can thus rely on a single `CDOSession` object to manage the user login session for all Cloud Data Services and their resources provided by a single Mobile or Web application. This single `CDOSession` object then manages the session life cycle from startup (login) to shutdown (logout) for all CDOs of an application and the Cloud Data Services they access from that same application.

Constructor

Instantiates a `CDOSession` object that you can use to start a user login session for a Mobile or Web application and load the CDO catalog for each supported Cloud Data Service whose resources are accessed using CDOs.

Syntax

```
CDOSession ( options )
```

options parameter

An object that has these properties:

serviceURI

The URI of the Mobile or Web application that the CDOSession will log into it must be an absolute URI.

authenticationModel (optional, defaults to Anonymous)

A string that specifies one of the three authentication models that the CDOSession supports:

CDOSession.AUTH_TYPE_ANON	"anonymous"
CDOSession.AUTH_BASIC_	"basic"
CDOSession.AUTH_TYPE_FORM	"form"

Properties

Member	Brief description (See also the reference entry)
authenticationModel property	A string constant that specifies the type of authentication that the server requires from the Mobile or Web App.
catalogURIs property	Returns the list of URIs used to load the CDO catalogs to access the Cloud Data Services provided by the Mobile or Web application for which the current CDO`Session` object manages a user login session.
clientContextId property	The value of the most recent client context identifier (CCID) that the `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.
CDOs property	Returns an array of CDOs that use the current `Session` object to communicate with their Data Object services.
connected property	Returns a `Boolean` that indicates the most recent online status of the current CDO`Session` object, when it last determined if the Mobile or Web application it manages is available.
loginHttpStatus property	Returns the specific HTTP status code returned in the response from the most recent login attempt on the current CDO`Session` object.

Member	Brief description (See also the reference entry)
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 CDO`Session` object.
onOpenRequest property	Returns the reference to a user-defined callback function that the CDO`Session` object executes to modify a request object before sending the request object to the server.
pingInterval property	A `Number` that specifies the duration, in milliseconds, between one automatic execution of the current CDO`Session` object's `ping( )` method and the next.
services property	Returns an array of objects that identifies the Cloud Data Services that have been loaded for the current CDO`Session` object and its Mobile or Web application.
serviceURI property	Returns the URI to the Mobile or Web application passed as a parameter to the constructor as a property of the constructor's *options* property.
userName property	Returns the user ID passed as a parameter to the most recent call to the `login( )` method on the current CDO`Session` object.

Methods

Session class-instance methods

Member	Brief description (See also the reference entry)
addCatalog( ) method	Loads a CDO catalog for a login session established using the `login( )` method.
login( ) method	Starts a user login session on the current CDO`Session` object by sending an HTTP request with user credentials to a URI for a specified Mobile or Web application.

Member	Brief description (See also the reference entry)
logout( ) method	Terminates the login session on the Mobile or Web application managed by the current CDO`Session` object, and invalidates any session currently maintained by the server.
ping( ) method	Determines the online status of the current CDO`Session` object from its ability to access the Mobile or Web application that it manages.
subscribe( ) method (Session class)	Subscribes a given event handler function to a named event of the current CDO`Session` object.
unsubscribe( ) method (Session class)	Unsubscribes a given event handler function from a named event of the current CDO`Session` object.

Events

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

Example — Using the CDOSession class

This is an example of how you might create a `CDOSession` object and use the URI to a Mobile or Web application to log into the application, load the CDO catalog for a Cloud Data Service provided by that application, and create a CDO for a `Customer` resource defined by that service in the catalog:

```
// create Session
pdsession = new CDOSession(serviceURI, authenticationModel );

// log in, i.e., authenticate to the Mobile Web application
pdsession.login(username, password, options);

// load catalog for a service that's part of the Mobile Web application
pdsession.addCatalog('https://BestSports.com:443/SportsApp/static/mobile/OrderEntrySvc.json');

// create CDO
customers = new progress.data.CDO( { name: 'Customer' } );

/* etc. - additional code to fill and use the CDO */
```

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

Notes

- 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 CDO catalogs for the session, and possibly call the `logout( )` method to terminate the session. To use the same `CDOSession` instance to start a new login session, you must call the `logout( )` method first.
- 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.

request Class

A request contains data and status information returned from a call to one of the methods of an associated `CDO` class instance (CDO) that executes a Data Object CRUD or invoke operation of a Cloud Data Service. This `request` object is returned by the associated CDO method call: `fill( )`, `saveChanges( )`, or a given invocation method.

In the case of an asynchronous call (all CRUD operations and invoke operations that you execute asynchronously), the `request` object is passed as a parameter to any user-defined event handler functions that you subscribe to associated CDO events. For invoke operations that you execute synchronously, the object is available as the return value of the corresponding CDO invocation method. The object is also passed as a parameter to any event handler functions that you subscribe to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO.

Properties

Member	Brief description (See also the reference entry)
async property	A `Boolean` that indicates, if set to `true`, that the Data Object operation was executed asynchronously on the Mobile or Web application.
batch property	A reference to an object with a property named `operations`, which is an array containing the request objects for each of the one or more Data Object record-change operations performed in response to calling the CDO `saveChanges( )` method either with an empty parameter list or with the single parameter value of `false`.
fnName property	For an invoke operation, the name of the CDO invocation method that called the operation.
cdo property	An object reference to the CDO that performed the operation returning the request object.
record property	An object reference to the record created, updated, or deleted by the current Data Object record-change operation.
objParam property	A reference to the object, if any, that was passed as an input parameter to the CDO method that has returned the current request object.
response property	Returns an object whose properties contain data from a Data Object built-in or invoke operation executed on the Cloud Data Server.
success property	A `Boolean` that when set to `true` indicates that the Data Object operation was successfully executed.
xhr property	A reference to the XMLHttpRequest object used to perform a Data Object operation request.

Methods

This object has no methods.

Events

This object has no events.

Cloud Data Object (CDO) Properties, Methods, and Events Reference

This section describes the properties, methods, and events of OpenEdge JavaScript classes and objects described in CDO Class and Object Reference. Where a given method can be called on different object type references, the method syntax shows how to call it for each object type and the description indicates when to call it and to what effect for each object type.

For details, see the following topics:

- caseSensitive property
- catalogURIs property
- cdo property
- CDOs property
- clearItems( ) method
- clientContextId property
- connected property
- data property
- deleteLocal( ) method
- display( ) method
- fill( ) method
- find( ) method
- findById( ) method
- fnName property
- foreach( ) method
- getData( ) method
- getErrorString( ) method
- getFormFields( ) method
- getFormRecord( ) method
- getId( ) method
- getListViewRecord( ) method
- getSchema( ) method
- hasData( ) method
- hasChanges( ) method
- invocation method
- record property
- records property
- login( ) method
- loginHttpStatus property
- loginResult property
- logout( ) method
- name property
- objParam property
- offline event
- online event

- ping( ) method
- pingInterval property
- readLocal( ) method
- record property
- rejectChanges( ) method
- rejectRowChanges( ) method
- remove( ) method
- response property
- saveChanges( ) method
- saveLocal( ) method
- services property
- serviceURI property
- setDetailPage( ) method
- setFieldTemplate( ) method
- setItemTemplate( ) method
- setListView( ) method
- setSortFields( ) method
- setSortFn( ) method
- showListView( ) method
- sort( ) method
- subscribe( ) method (CDO class)
- subscribe( ) method (Session class)
- success property
- table reference property (CDO class)
- unsubscribe( ) method (CDO class)
- unsubscribe( ) method (Session class)
- unsubscribeAll( ) method
- update( ) method
- useRelationships property
- userName property
- xhr property

_errorString property (deprecated)

NOTE: _errorString property has been deprecated. Use getErrorString() method instead.

A string value available on the `data` property of every `CloudDataRecord object` in CDO memory that is set as part of before-image data for the record and provides descriptive information about any record change error on the server following a Data Object create, update, delete, or submit operation.

If there is no error in the associated record change, the value of this property is `undefined`. However, this property can have a value when a given record change involving the associated record object fails. For an OpenEdge Cloud Data Service, the value of this property then corresponds to the setting of the `ERROR-STRING` attribute on the associated temp-table buffer object on the Cloud Data Server.

Note: To return this value to your application for any record object in the CDO, use the `getErrorString( )` method.

Return type: `String`

Access: Read-only

Applies to: data property

You can access this value on the `data` property of a `CloudDataRecord object` that is returned from a record change operation and obtained in one of the following ways for a CDO that supports before-imaging:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `records` property on the request object returned to the callback of a CDO `afterSaveChanges` event on completion of a Data Object submit operation.

Example:

Following are valid `_errorString` property references, where `record-ref` is a valid `CloudDataRecord object` reference, `dsCustomer` is a CDO reference, and `ttCustomer` is a table reference in the CDO:

```
record-ref.data._errorString
dsCustomer.ttCustomer.record.data._errorString
```

_id property (deprecated)

NOTE: _id property has been deprecated. Use getId() method instead.

A string value available on the `data` property of every `CloudDataRecord object` in CDO memory that provides a unique internal ID for the record.

This internal record ID is a unique value generated by OpenEdge for each `CloudDataRecord` record object loaded in CDO memory using the `fill( )`, `add( )`, or `addRecords( )` methods. This field has no relationship to the internal `RECID` and `ROWID` values maintained for the records of an OpenEdge database. Use this record ID to relate records in a table hierarchy.

Note: To return this value to your application for any record object in the CDO, use the `getId( )`

method.

Data type: `String`

Access: Read-only

To return and set the specified record as the working record, you can pass any `_id` value to the

`findById( )` method called on the associated table reference.

Note: The value assigned to `_id` for any given record object can change with each invocation of the `fill( )` or `saveChanges( )` methods.

Caution: Do not change the value referenced by `_id`. Otherwise, any Mobile or Web application UI managed by OpenEdge can have unpredictable behavior.

Examples

Following are valid `_id` property references, where `record-ref` is a valid `CloudDataRecord object` reference, `dsCustomer` us a CDO reference, and `ttCustomer` is a table reference in the CDO:

```
record-ref.data._id
dsCustomer.ttCustomer.record.data._id
```

acceptChanges( ) method

Accepts changes to the data in CDO memory for the specified table reference or for all table references of the specified CDO.

If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method applies only when the CDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method **after** calling the `saveChanges( )` method in order to accept a series of changes after they have been successfully applied to the Cloud Data Server. If the `autoApplyChanges` property is `true`, the CDO automatically accepts or rejects changes for the specified table reference, or for all table references of the specified CDO, based on the success of the corresponding Data Object record-change operations.

Note: Accepting all pending changes in CDO memory—or even pending changes for a single table reference—because none raised an error from the Cloud Data Server might be too broad an action for your application. If so, consider using `acceptRowChanges( )` to accept changes to a single table record at a time. For more information, see the description of `acceptRowChanges( )` method.

Return type: `Boolean`

Syntax

```
cdo-ref.acceptChanges ( )
cdo-ref.table-ref.acceptChanges ( )
```

`cdo-ref`

A reference to the CDO. If you call the method on *`cdo-ref`*, the method accepts changes for all table references in the CDO.

`table-ref`

A table reference on the CDO. If you call the method on *`table-ref`*, the method accepts changes for the specified table reference.

When you accept changes on a table reference, this method makes the record objects for the specified table reflect all pending changes in CDO memory. When you accept changes on the CDO reference, the method makes the record objects for all table references in the CDO reflect all pending changes in CDO memory. As the specified changes are accepted, the method also empties any associated before-image data, clears all associated settings of the `getErrorString() method`, and removes the associated record change indications from CDO memory.

Note: After this method accepts changes, and if you have set up automatic sorting using the `autoSort` property, all the record objects for affected table references are sorted accordingly. If the sorting is done using sort fields, any `String` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have pending CDO changes that you need to apply to the Cloud Data Server, be sure **not** to invoke this method **before** you invoke the `saveChanges( )` method to successfully apply these changes to the Cloud Data Server. Otherwise, the affected client data will be inconsistent with the corresponding data on the Cloud Data Server.

Example

The following code fragment shows a CDO created so it **does not** automatically accept or reject changes to data in CDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a handler for the CDO `afterSaveChanges` event to determine if all changes to the `eCustomer` table in CDO memory should be accepted or rejected based on the success of all Data Object create, update, and delete operations on the Cloud Data Server. If the success parameter is false, one or more of the rows was returned with an error. To change the data for a record, a jQuery event is also defined on an update button to update the corresponding `eCustomer` record in CDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.CDO( { name: 'dsCustomerOrder',
                                   autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterSaveChanges', onAfterSaveCustomers, this);

$('#btnUpdate').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.assign();
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnSave').bind('click', function(event) {
  dataSet.saveChanges();
});

function onAfterSaveCustomers(cdo, success, request) {
  if (success)
  {
     cdo.eCustomer.acceptChanges();
     // Additional actions associated with accepting the pending changes...
  }
  else
  {
     // At least one of the row changes failed so want to reject all changes.
     //Additional actions associated with rejecting the pending changes...
    cdo.eCustomer.rejectChanges();
  }
}
```

When the update button is clicked, the event handler uses the `findById( )` method to find the original record (`record`) with the matching internal record ID (`#id`) and invokes the `assign( )` method on `record` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete `eCustomer` records and add new `eCustomer` records.

A jQuery event also defines a save button that when clicked invokes the `saveChanges( )` method to apply all pending changes in CDO memory to the Cloud Data Server. After the method completes, and all results have been returned to the client from the Cloud Data Server, the CDO `afterSaveChanges` event fires, and if all Data Object operations on the Cloud Data Server were successful, the handler calls `acceptChanges( )` to accept the pending changes to `eCustomer` in CDO memory. For more information on how this same example determines when and how to reject changes, see the description of the `rejectChanges( )` method.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each Data Object record-change operation, one record at a time, across the network. You can also have `saveChanges( )` send all pending record change operations across the network in a single Data Object submit operation. For more information and an example, see the description of the `saveChanges( )` method.

acceptRowChanges( ) method

Accepts changes to the data in CDO memory for a specified record object.

This can be the working record of a table reference or the record specified by a `CloudDataRecord object` reference. If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method applies only when the CDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method for a successful Data Object record-change operation in the handler for the corresponding CDO event fired in response to executing the `saveChanges( )` method. If the `autoApplyChanges` property is `true`, the CDO automatically accepts or rejects changes to the record object based on the success of the corresponding Data Object operation on the Cloud Data Server.

Return type: `Boolean`

Syntax

```
record-ref.acceptRowChanges ( )
cdo-ref.acceptRowChanges ( )
cdo-ref.table-ref.acceptRowChanges ( )
```

`record-ref`

A reference to a `CloudDataRecord object` for a table reference in

CDO memory. You can obtain a `CloudDataRecord object` by:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`, `afterDelete`, or `afterDelete` event.

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the CDO that has a working record.

When you accept changes on a specified record object, this method makes the record reflect all pending changes in CDO memory. As the specified changes are accepted, the method also empties any associated before-image data, clears any associated settings for the getErrorString() methodand removes the associated pending change indications from CDO memory.

Note: After this method accepts changes on a record, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `String` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have pending CDO changes that you need to apply to the Cloud Data Server, be sure **not** to invoke this method **before** you invoke the `saveChanges( )` method. Otherwise, the affected client data will be inconsistent with the corresponding data on the Cloud Data Server.

Example

The following code fragment shows a CDO created so it **does not** automatically accept or reject changes to data in CDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single handler for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in CDO memory should be accepted or rejected based on the success of the corresponding Data Object operation on the Cloud Data Server. To change the data for a record, a jQuery event is also defined on a save button to update the corresponding `eCustomer` record in CDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.CDO( { name: 'dsCustomerOrder',
                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnSave').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.assign();
  dataSet.saveChanges();
});

// Similar controls might be defined to delete and create eCustomer records...

function onAfterCustomerChange(cdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
    // Perform other actions associated with accepting this record change
  }
  else
  {
    record.rejectRowChanges();
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `record` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. It then calls the `saveChanges( )` method to invoke the Data Object update operation to apply these record changes to the Cloud Data Server. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

After each Data Object operation for a changed `eCustomer` record completes, results of the operation are returned to the client from the Cloud Data Server, and the appropriate event fires. If the operation was successful, the handler calls `acceptRowChanges( )` to accept the record change associated with the event in CDO memory. An advantage of using an event to manually accept a record change is that you can perform other actions associated with accepting this particular change, such as creating a local log that describes the change.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each Data Object operation, one record at a time, across the network. You can also have `saveChanges( )` send all pending record change operations across the network in a single Data Object submit operation. For an example, see the description of the `saveChanges( )` method.

add( ) method (Same as create() method)

Creates a new record object for a table referenced in CDO memory and returns a reference to the new record.

After completing execution, the new record becomes the working record for the associated table. If the table has child tables, the working record for these child tables is not set. To synchronize the change on the Cloud Data Server, call the `saveChanges( )` method.

Return type: CloudDataRecord object

Syntax

```
cdo-ref.add ( [new-record-object] )
cdo-ref.table-ref.add ( [new-record-object] )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`new-record-object`

If specified as a non-`null` object, passes in the data to create the record for the `CloudDataRecord instance`. The data to create the record is identified by one or more properties, each of which has the name of a corresponding field in the table schema and has the value to set that field in the new table record.

If you omit or set the parameter to `null`, or you do not include properties of

`new-record-object` for all fields in the new record, the method uses the default values from the table schema stored in the catalog to set the unspecified record fields.

Note: After this method adds the new record object, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `String` fields are compared according to the value of the `caseSensitive` property.

If the specified table reference is for a child table in a ProDataSet, when the `useRelationships` property is `true`, `add( )` uses the relationship to set related field values of the new child record from the working record of the parent table. However, if the working record of the parent is not set, `add( )` throws an error. If `useRelationships` is `false`, the fields for the new record are set as specified by *`new-record-object`* and no error is thrown.

Example

Assuming `useRelationships` is `true`, given a CDO created for a ProDataSet resource with a `customer` and related child `order` table, the `add( )` method in the following code fragment uses this relationship to automatically set the `CustNum` field in a new record added to the `order` table:

```
var dataSet = new Progress.data.CDO( 'CustomerOrderDS' );
dataSet.customer.add( { CustNum: 1000, Balance: 10000, State: 'MA' } );

// CustNum is set automatically by using the relationship
dataSet.order.add( { OrderNum: 1000 } );
```

Note: OpenEdge adds the new record object with an OpenEdge-reserved `String` returned by getId(), which uniquely identifies the record in CDO memory. Note that once you have saved the new record object to the Cloud Data Server using `saveChanges( )`, this value can change with each invocation of the `fill( )` method.

addCatalog method

Obtains and processes a CDO catalog. The appropriate catalog needs to be loaded before creating a CDO for the resource(s) defined in the catalog. This method will throw an error if it is not possible to send a request to the Web application.

Return type: `undefined or a Concurrency object`

Syntax

```
addCatalog( catalogURI [ , username, password] [ , options ] )

addCatalog( catalogURIs[] [ , username, password ] [ , options] )
```

catalogURI parameter

The URI of a CDO catalog that the call will retrieve and process. The URI will be used exactly as it is passed to the addCatalog() method. This means that if it is a relative path (i.e., it does not begin with a scheme (protocol) or a single slash), the catalog location will be treated as being local to the client app (that is, relative to the location of the document that loaded the file that contains the CDOSession code).

catalogURIs parameter
An array of strings, each of which is the URI for a catalog as defined above under the *catalogURI* parameter. All of the catalogs in the array must be accessible without specifying credentials (because they are unprotected, or because the CDOSession has already logged in to the Web application where they are located), or by using the credentials passed to this call plus the authenticationModel of the CDOSession object. .

userName
Ignored if using Anonymous authentication, necessary otherwise. If used in the addCatalog signature that takes an array of catalog URIs, the userName will be applied to all of them.

password
Ignored if using Anonymous authentication, necessary otherwise. If used in the addCatalog signature that takes an array of catalog URIs, the password will be applied to all of them.

options
An object that has the following property:

iOSBasicAuthTimeout (optional)

This setting applies only if the addCatalog is an asynchronous request using Basic authentication made from an iOS device. The value is the time, in milliseconds, that the addCatalog() method will wait for a response before generating an error (an error may mean that invalid credentials were sent). If this value is 0, no timeout will be set. If the iOSBasicAuthTimeout is not present, addCatalog() will use the default of 4 seconds. (This property, and its implementation, is a workaround for a bug in Apache Cordova that may affect asynchronous requests sent from iOS devices using HTTP Basic authentication where the credentials are incorrect. Such requests have been found to hang rather than correctly call the response handler with a 401 Unauthorized status code.)

The addCatalog method executes asynchronously. It may be implemented to communicate the result by firing an event named *afterAddCatalog* or through the use of a concurrency object, for example a promise object in JavaScript. The signatures for *afterAddCatalog* or for handlers associated with a concurrency object are the same:

session
Reference to the CDOSession object on which addCatalog() was called

result
The overall result of the call. There are two possible values:
progress.data.Session.SUCCESS
progress.data.Session.GENERAL_FAILURE
The value will be SUCCESS if each catalog specified by the catalogURI(s) in the call was retrieved successfully or had already been loaded. Otherwise, the result will be GENERAL_FAILURE.

details

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

catalogURI

result :

progress.data.Session.SUCCESS

catalog load succeeded

progress.data.Session.AUTHENTICATION_FAILURE

authentication error on the attempt to load the catalog

progress.data.Session.GENERAL_FAILURE

other error on the attempt to load the catalog

progress.data.Session.CATALOG_ALREADY_LOADED

the catalog had been loaded previously

errorObject

any error object thrown while adding the catalog

xhr

Reference to the XMLHttpRequest object used to make the addCatalog request

Note that when addCatalog() is requested to load a catalog that is already loaded, it does not load it again. This is considered to be a successful execution.

Note that addCatalog may be called even when there has been no successful login.

addLocalRecords( ) method

Reads the record objects stored in the specified local storage area and updates CDO memory based on these record objects, including any pending changes and before-image data, if they exist.

The method updates any single array or table, or all tables for a ProDataSet, as read in according to the CDO resource definition. The data is merged into CDO memory and affects existing data according to a specified merge mode and optional key fields.

After execution, the working record set for each CDO table reference remains unchanged.

Return type: `Boolean`

Syntax

```
addLocalRecords ( [ storage-name , ] add-mode [ , key-fields ] )
```

`storage-name`

The name of the local storage area in which to save the specified data from CDO memory. If *`storage-name`* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `cdo_`*`serviceName`*`_`*`resourceName`*, where *`serviceName`* is the name of the Cloud Data Service that supports the CDO instance, and `resourceName` is the name of the resource (table, dataset, etc.) for which the CDO instance is created.

Note: A ProDataSet object read in from local storage can contain before-image data, which this method merges into CDO memory along with the record objects. However, if the ProDataSet object contains before-image data for a record object that conflicts with existing before-image data in CDO memory for that same record object, `addLocalRecords( )` throws an exception.

`add-mode`

An integer constant that represents a merge mode to use. Each merge mode handles duplicate keys in a particular manner, depending on your specification of *`key-fields`*. You can specify the following numeric constants, which affect how the table record objects in the specified local storage area are added to CDO memory:

- **`MODE_REPLACE`** — Adds the table record objects in the specified local storage area to the existing record objects in CDO memory. If duplicate keys are found between record objects in local storage and record objects in CDO memory, the record objects with duplicate keys in CDO memory are replaced with the corresponding records in local storage.

  Note: For the current release, only this single merge mode is supported. Use of any other merge mode (for example, as specified for the `addRecords( )` method) throws an exception.

Caution:

If any specified *`key-fields`* match the unique indexes of corresponding tables on the server, adding the contents of the specified local storage area can result in records with duplicate keys. If the corresponding server tables have unique indexes, you must make any affected duplicate key fields unique before calling `saveChanges( )`.

`key-fields`

An object with a list of primary key fields to check for records with duplicate keys. For example, when merging with a ProDataSet that has `eCustomer` and `eOrder` table references, you might use the following object:

```
{
  eCustomer: [ "CustNum" ],
  eOrder: [ "CustNum", "Ordernum" ]
}
```

When merging with a single table reference, you might use the following array object:

```
[ "CustNum", "Ordernum" ]
```

Note: For any *key-fields* that have the String data type, the character values for these fields are compared to identify duplicates according to the value of the caseSensitive property on each affected table reference.

If *key-fields* is specified, the method checks for duplicate keys using the specified primary keys found in *key-fields*. If *key-fields* is **not** specified, the method searches other possible sources for definitions of primary keys in the following order, and uses the first source of definitions found:

1. Primary key annotations from any OpenEdge Business Entity resource (as identified in the CDO Catalog)
2. Unique ID properties associated with the resource (for example, the idProperty property as identified in the CDO Catalog for a Rollbase object)

If no source of primary key definitions is found, the method adds **all** local storage records to CDO memory, regardless of the specified *add-mode*, and regardless of any duplicate records that might result.

Note: After this method checks for any duplicate keys and completes adding record objects to CDO memory, and if you have set up automatic sorting using the autoSort property, all the record objects for the affected table references are sorted accordingly. If the sorting is done using sort fields, any String values in the specified sort fields are compared according to the value of the caseSensitive property.

This method returns true if it successfully reads the data from the local storage area; it then updates CDO memory with this data according to the specified *add-mode*. If *storage-name* does not exist, but otherwise encounters no errors, the method leaves CDO memory unchanged and returns false. If the method does encounter errors (for example, with reading the data in the specified storage area), it also leaves CDO memory unchanged and throws an exception.

Example

The following code fragment fills memory for a CDO, dataset, with records from a csCustomerOrder ProDataSet on the server. This ProDataSet contains temp-tables that correspond to the Customer and Order tables of the OpenEdge sports2000 database:

```
var dataset = progress.data.CDO( "dsCustomerOrder" );
dataset.fill(); // Loads the ProDataSet with all available server records

// Adds records
dataset.addLocalRecords( progress.data.CDO.MODE_REPLACE, [ "CustNum",
"Ordernum" ] );
```

The fragment then calls addLocalRecords( ) on the CDO to add a set of similar records to CDO memory from the default local storage area, where the records were previously stored using the CDO saveLocal( ) method. Duplicate Customer and Order records are checked and replaced with the records from local storage based on the respective primary key fields, CustNum and Ordernum.

addRecords( ) method

Reads an array, table, or ProDataSet object containing one or more record objects and updates CDO memory based on these record objects, including any pending changes and before-image data, if they exist.

The method updates all tables read in for a ProDataSet or updates a specified CDO table, depending on how the method is called. The data is merged into CDO memory and affects existing data according to a specified merge mode and optional key fields.

After execution, the working record set for each CDO table reference depends on the merge mode that is specified.

Return type: `null`

Syntax

```
cdo-ref.addRecords ( merge-object , add-mode [ , key-fields ] )
cdo-ref.table-ref.addRecords ( merge-object , add-mode [ , key-fields ] )
```

`cdo-ref`

A reference to the CDO. If you call the method on *`cdo-ref`*, the method merges data for all referenced tables in the ProDataSet.

`table-ref`

A table reference on the CDO. If you call the method on *`table-ref`*, the method merges data only for the referenced table.

`merge-object`

An object with the data to merge. If you call the method on *`table-ref`*, the object can either be an object that contains an array of record objects to merge with the referenced table or a ProDataSet-formatted object containing such an array.

Note: This object must have a supported JavaScript object format that matches the data returned from the built-in read operation (CDO `fill( )` method). For example, the object returned from an invocation method for an output table or ProDataSet that has the same schema as supported output from the built-in read operation should work.

The following formats are supported for *`merge-object`*:

- Single table object with an array of record objects. For example:

```
{
  eCustomer: [
    // Record objects ...
  ]
}
```

- An array of record objects for a single table object or for a ProDataSet with a single table object. For example:

```
[
  // Record objects ...
]
```

- A ProDataSet object with a single table or multiple table objects at the same level only. For example:

```
{
  dsCustomerOrder: {
    eCustomer: [
      // Record objects ...
    ],
    eOrder: [
      // Record objects ...
    ]
  }
}
```

Note: A ProDataSet object can contain before-image data, which this method merges into CDO memory along with the record objects. However, if the ProDataSet object contains before-image data for a record object that conflicts with existing before-image data in CDO memory for that same record object, `addRecords( )` throws an exception.

add-mode

An integer that represents a merge mode to use. If you also specify *key-fields*, each merge mode handles duplicate keys in a particular manner as described here. If you **do not** specify *key-fields*, the method adds **all** the records of *merge-object* regardless of the mode. You can specify the following numeric constants, which affect how the table record objects in *merge-object* are added to CDO memory:

- **MODE_APPEND** — Adds the table record objects in *merge-object* to the existing record objects in CDO memory. If a duplicate key is found between a record object in *merge-object* and a record object in CDO memory, the method throws an error.
- **MODE_MERGE** — Adds the table record objects in *merge-object* to the existing record objects in CDO memory. If duplicate keys are found between record objects in *merge-object* and record objects in CDO

memory, the method ignores (does not add) the record objects with duplicate keys in merge-object.

- **`MODE_REPLACE`** — Adds the table record objects in *`merge-object`* to the existing record objects in CDO memory. If duplicate keys are found between record objects in *`merge-object`* and record objects in CDO memory, the record objects with duplicate keys in CDO memory are replaced with the corresponding records in *`merge-object`*.
- **`MODE_EMPTY`** — Empties all table record objects from CDO memory and replaces them with the contents of *`merge-object`*.

  Note: If *`merge-object`* is an empty object (`{}`), this mode effectively empties the data from CDO memory.

After execution, if the specified merge mode was `CDO.MODE_EMPTY`, the working record set for any table references is `undefined`, because CDO memory is completely emptied or replaced. For any other merge mode, the working record set for each CDO table reference remains unchanged.

Caution:

If a table's *`key-fields`* matches the unique indexes of corresponding tables, adding the contents of *`merge-object`* can result in records with duplicate keys. If the corresponding tables have unique indexes, you must make any affected duplicate key fields unique before calling `saveChanges( )`.

`key-fields`

An object with a list of key fields to check for records with duplicate keys. For example, when merging with a ProDataSet that has `eCustomer` and `eOrder` table references, you might use the following object:

```
{
  eCustomer: [ "CustNum" ],
  eOrder: [ "CustNum", "Ordernum" ]
}
```

When merging with a single table reference, you might use the following array object:

```
[ "CustNum", "Ordernum" ]
```

Note: For any *`key-fields`* that have the `String` data type, the character values for these fields are compared to identify duplicates according to the value of the `caseSensitive` property on each affected table reference.

Note: After this method checks for any duplicate keys and completes adding record objects to CDO memory from *merge-object*, and if you have set up automatic sorting using the autoSort property, all the record objects for the affected table references are sorted accordingly. If the sorting is done using sort fields, any String values in the specified sort fields are compared according to the value of the caseSensitive property.

A typical use for addRecords( ) is to merge additional data returned by an invocation method without having to re-load CDO memory with all the data from the fill( ) method.

Example

Given a CDO, dataset, that you fill with available records from the eCustomer and eOrder tables, you might retrieve a new eOrder record as the result of a getNewOrder( ) invocation method on the CDO and add the new record to CDO memory as follows:

```
var dataset = progress.data.CDO( "dsCustomerOrder" );
dataset.fill(); // Loads the ProDataSet with all available records

// Adds a new eOrder record restrieved from the service
var request = dataset.getNewOrder(null,false);
dataset.eOrder.addRecords( request.response, progress.data.CDO.MODE_APPEND,

  [ "CustNum", "Ordernum" ],
  );
```

This code fragment adds the eOrder record for an existing eCustomer record specified by the CustNum property and a new order number specified by the Ordernum property of the single record object returned in result.dsCustomerOrder.eOrder[0].

afterAddCatalog event

Fires when the addCatalog( ) method on the current CDOSession object completes execution.

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

Syntax

```
function ( login-session , result , details
 )
```

login-session

A reference to the CDOSession object that fired the event.

result

A numeric constant that the addCatalog( ) method returns when called synchronously. Possible constants include:

- CDO**Session.SUCCESS** — The specified CDO catalog loaded successfully.
- CDO**Session.AUTHENTICATION_FAILURE** — The catalog failed to load because of a user authentication error.

- CDO**Session.CATALOG_ALREADY_LOADED** — The specified CDO catalog did not load because it is already loaded

For all other errors, this event returns an Error object reference as *error-object*. For more detailed information about any response (successful or unsuccessful) returned from the Web server, you can also check the XMLHttpRequest object (XHR) returned by the lastSessionXHR property.

If *error-object* is **not** null, *result* will be null.

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

catalogURI
result :
progress.data.Session.SUCCESS
catalog load succeeded
progress.data.Session.AUTHENTICATION_FAILURE
authentication error on the attempt to load the catalog
progress.data.Session.GENERAL_FAILURE
other error on the attempt to load the catalog
progress.data.Session.CATALOG_ALREADY_LOADED
the catalog had been loaded previously

errorObject
any error object thrown while adding the catalog
xhr
Reference to the XMLHttpRequest object used to make the addCatalog request

Application code can subscribe a handler to this event by invoking the subscribe( ) method on a CDOSession object.

Example

The following code fragment subscribes the function, onAfterAddCatalog, to handle the afterAddCatalog event fired on the session, empSession, after the addCatalog( ) method is called. The event handler checks for either expected success and failure return values, or a thrown Error object with an unknown error, and assembles an appropriate message to display in an alert box for each case:

```
var retValue;
empSession.subscribe('afterAddCatalog', onAfterAddCatalog);

retValue = empSession.addCatalog(myCatalogURI);
/* ( retValue is progress.data.Session.ASYNC_PENDING ) */


/* invoked by empSession when it processes the response from
   getting the catalogfrom the Web application */
function onAfterAddCatalog( pdsession, addCatalogResult, details) {
    var msg;

    if (addCatalogResult ===
          CDOSession.SUCCESSCDOSession.LOGIN_AUTHENTICATION_FAILURE ) {
        alert("Catalogs loaded.");
    }
      /* only 1 catalog was requested, so checking just the 1st
         element of the details array */
    else if (details[0].result ===
        CDOSession.LOGIN_AUTHENTIATION_FAILURE ) {
    }
    else {
        if (errorObject) {
            msg = '\n' + errorObject.message;
        }
        alert("unexpected addCatalog error." + msg);
    }
}
```

afterCreate event

Fires after the CDO, by means of a `saveChanges( )` call following an `add( )` or create call(), sends a request to create a record and receives a response to this request from the Cloud Data Server.

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

Syntax

```
function ( cdo , record , success , request )
```

`cdo`

A reference to the CDO that invoked the create operation. For more information, see the description of cdo property of the request object.

`record`

A reference to the table record upon which the create operation acted. For more information, see the description of record property of the request object.

`success`

A `Boolean` that is `true` if the create operation was successful. For more information, see the description of success property of the request object.

request

A reference to the request object returned after the create operation completes. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onAfterCreate`, to handle the `afterCreate` event fired on the single-table CDO, `mycdo`, where `newDataObject` is an object containing the field values to assign in the new record:

```
/* subscribe to event */
mycdo.subscribe( 'afterCreate', onAfterCreate );

/* some code that would add a record and save it */
var record = mycdo.add( newDataObject );

. . .

mycdo.saveChanges();

function onAfterCreate ( cdo , record , success , request ) {
    var jsrecError;
    if (success) {

        /* for example, get the values from the record for redisplay */
        var myField = record.data.myField;
        . . .
    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */

            }
        }

        /* Call getErrorString() on request.jsrecord to return any
         * record-change row error in before-image data (if present)
         * and handle the error */
        jsrecError = record.getErrorString();
        if (jsrecError) {
            /* process current record-change error */
        }
    }
};
```

afterDelete event

Fires after the CDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record and receives a response to this request from the Cloud Data Server.

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

Syntax

```
function ( cdo , record , success , request )
```

cdo

A reference to the CDO that invoked the delete operation. For more information, see the description of cdo property of the request object.

record

A reference to the table record upon which the delete operation acted. For more information, see the description of record property of the request object.

success

A `Boolean` that is `true` if the delete operation was successful. For more information, see the description of success property of the request object.

request

A reference to the request object returned after the delete operation completes. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onAfterDelete`, to handle the `afterDelete` event fired on the CDO, `mycdo`, where `myid` is the known ID of a record to find and delete:

```
/* subscribe to event */
mycdo.subscribe( 'afterDelete', onAfterDelete );

/* some code that would delete a record and send to the server */
var record = mycdo.findById(myid);
record.remove();
mycdo.saveChanges();

function onAfterDelete ( cdo , record , success , request ) {
    var jsrecError;
    if (success) {

        /* for example, get the values from the record that was
           deleted to display a confirmation message */
        var myKeyField = record.data.myKeyField;
        . . .
    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */

            }
        }

        /* Call getErrorString() on request.jsrecord to return any
         * record-change row error in before-image data (if present)
         * and handle the error */
        jsrecError = record.getErrorString();
        if (jsrecError) {
            /* process current record-change error */
        }

    }
};
```

afterFill event

Fires after the CDO, by means of a `fill( )` call, sends a request to read a table or ProDataSet into CDO memory and receives a response to this request from the Cloud Data Server.

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

Syntax

```
function ( cdo , success , request )
```

cdo

A reference to the CDO that invoked the fill operation. For more information, see the description of cdo property of the request object.

success

A Boolean that is true if the fill operation was successful. For more information, see the description of success property of the request object.

request

A reference to the request object returned after the fill operation completes. For more information, see the description of request object.

Example

The following code fragment subscribes the function, onAfterFill, to handle the afterFill event fired on the CDO, mycdo:

```
mycdo.subscribe( 'afterFill', onAfterFill );
mycdo.fill();

function onAfterFill( cdo , success , request ) {
    if (success) {

        /* for example, add code to display all records on a list */
        cdo.foreach(function (record) {
            /* you can reference the fields as record.data.field */
        });
    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */

            }
        }
    }

};
```

afterInvoke event

Fires after a non-built-in method is called asynchronously on a CDO and a response to the request is received from the Cloud Data Server.

Synchronous method calls do not cause this event to fire.

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

Syntax

```
function ( cdo , success , request )
```

cdo

A reference to the CDO that invoked the method. For more information, see the description of cdo property of the request object.

success

A `Boolean` that is `true` if the operation was successful. For more information, see the description of success property of the request object.

request

A reference to the request object returned after the operation completes. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onAfterInvokeMyMethod`, to handle the `afterInvoke` event fired on the CDO, `mycdo`, for an invocation of the `myMethod( )` invocation method passed the parameters specified by `paramObject`:

```
mycdo.subscribe( 'afterInvoke', 'myMethod', onAfterInvokeMyMethod );
mycdo.myMethod( paramObject );

function onAfterInvokeMyMethod( cdo , success , request )
    if (success) {

        var response = request.result.response;
        var retval = response._retval;
        var myOutputParm = response.myOutParam;

    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */

            }
        }
    }

};
```

afterLogin event

Fires when the `login( )` method on the current `CDOSession` object completes execution.

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

Syntax

```
function ( login-session , result , info )
```

`login-session`

A reference to the `Session` object that fired the event.

`result`

A numeric constant that the `login( )` method returns when called synchronously. Possible constants include:

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

For all other errors, this event returns an `Error` object reference in the info parameter. You can also return the result for the most recent login attempt on *`login-session`* by reading its `loginResult` property. For a more specific status code returned in the HTTP response, you can check the value of its `loginHttpStatus` property. For more detailed information about any response (successful or unsuccessful) returned from the Web server, you can also check the XMLHttpRequest object in the xhr property in the info parameter..

If *`error-object`* is **not** `null`, *`result`* will be `null`.

`info`

a JavaScript object that can have the following properties:

errorObject

A reference to any Error object thrown during the login

xhr

Reference to the XMLHttpRequest object used to make the login request to the Web application

Application code can subscribe a handler to this event by invoking the `subscribe( )` method on a `CDOSession` object.

Example

The following code fragment subscribes the function, `onAfterLogin`, to handle the `afterLogin` event fired on the session, `empSession`, after the `login( )` method is called asynchronously. The event handler checks for either an expected return value, an invalid return value (as part of a test), or a thrown `Error` object with an unknown error (passed as `errorObject` to the handler), then assembles an appropriate message to display in an alert box:

```
var retValue;
empSession.subscribe('afterLogin', onAfterLogin);

retValue = empSession.login( { serviceURI : serviceURI,
                               userName : uname,
                               password : pw,
                               async : true }  );
/* ( retValue is progress.data.Session.ASYNC_PENDING ) */

/* Invoked by empSession when it processes the login response from the Web
   application */
function onAfterLogin( pdsession, result, info ) {
    var msg;
    var loginResult = pdsession.loginResult;

    if ( loginResult === null ) {
        msg = "Employee Login failed. Error attempting to call login";
    }
    else if ( loginResult ===
progress.data.Session.LOGIN_AUTHENTICATION_FAILURE ) {
            msg = "Employee Login failed. Authentication error";
    }
    else if ( loginResult === progress.data.Session.LOGIN_GENERAL_FAILURE  )
 {
        msg = "Employee Login failed. Unspecified error";
    }
    else  if ( loginResult === progress.data.Session.LOGIN_SUCCESS  ) {
        msg = "Logged in successfully";
    }
    else {
           if (info.errorObject) {
               msg = '\n' + info.errorObject.message;
           }
        msg = "TEST ERROR! UNEXPECTED loginResult" + msg;
    }
    msg = msg +
            "\nloginResult: " + pdsession.loginResult +
            "\nloginHttpStatus: " + pdsession.loginHttpStatus +
            "\nuserName: " +  pdsession.userName +
            "\nlastSessionXHR: " +  pdsession.lastSessionXHR;

    alert(msg);
}
```

afterLogout event

Fires when the `logout( )` method on the current `Session` object completes execution.

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

Syntax

```
function ( login-session , result, info)
```

login-session

A reference to the `Session` object that fired the event.

result
Indicates outcome of logout:

progress.data.Session. SUCCESS the logout succeeded
progress.data.Session.GENERAL_FAILURE some error on the logout attempt

info
a JavaScript object that can have the following properties:

errorObject	A reference to any Error object thrown during the logout
xhr	Reference to the XMLHttpRequest object used to make the logout request to the Web application

Note: The `logout( )` method does not send a request to the Web application if it is using Anonymous authentication. In this case, `logout( )` will nevertheless invoke any `afterLogout` event handler that has been subscribed when it is done executing.

Application code can subscribe a handler to this event by invoking the `subscribe( )` method on a `CDOSession` object.

Example

The following code fragment subscribes the function, `onAfterLogout`, to handle the `afterLogout` event fired on the session, `empSession`, after the `logout( )` method is called asynchronously. If an `Error` object is passed in, the event handler displays a message:

```
empSession.subscribe('afterLogout', onAfterLogout);

empSession.logout( { async : true }  );


/* Invoked by empSession when it finishes executing the logout operation */
function onAfterLogout( pdsession, result, info ) {
    var msg;

    msg = info.errorObject ? '\n' + info.errorObject.message : '';
    if ( pdsession.lastSessionXHR === null ) {
        alert("logout succeeded");
        return;
    }
    alert("There was an error attempting to log out." + msg);
}
```

afterRead event

Fires after the CDO, by means of a `read( )` call, sends a request to read a table or ProDataSet into CDO memory and receives a response to this request from the Cloud Data Server.

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

Syntax

```
function ( cdo , success , request )
```

`cdo`

A reference to the CDO that invoked the read operation. For more information, see the description of cdo property of the request object.

`success`

A `Boolean` that is `true` if the read operation was successful. For more information, see the description of success property of the request object.

`request`

A reference to the request object returned after the read operation completes. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onAfterRead`, to handle the `afterRead` event fired on the CDO, `mycdo`:

```
mycdo.subscribe( 'afterRead', onAfterRead );
mycdo.fill();

function onAfterRead( cdo , success , request ) {
    if (success) {
        /* for example, add code to display all records on a list */
        cdo.foreach(function (record) {
            /* you can reference the fields as record.data.field */
        });
    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {
            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */

            }
        }
    }

};
```

afterSaveChanges event

Fires once for each call to the `saveChanges( )` method on a CDO, after responses to all create, update, and delete requests have been received from the Cloud Data Server.

The signature of the event handler function:

Syntax

```
function ( cdo , success , request )
```

`cdo`

A reference to the CDO that invoked the `saveChanges( )` method. For more information, see the description of cdo property of the request object.

`success`

A `Boolean` that is `true` if all operations initiated by `saveChanges( )` were successful, and false if at least one of the operations failed. For more information, see the description of success property of the request object.

`request`

A reference to the request object returned after all requested operations complete. For more information, see the description of request object.

Example

The following code fragment subscribes the function, onAfterSaveChanges, to handle the

afterSaveChanges event fired on the CDO, mycdo, where saveChanges() without submit functionality is called:

```
/* subscribe to event */
mycdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );

/* some code that would do multiple CRUD operations and
   send them to the server */
var newrec = mycdo.add();

. . .

var record = mycdo.findById(myid);
record.remove();
mycdo.saveChanges();
function onAfterSaveChanges( cdo ,   success , request ) {

    /* number of operations on batch */
    var len = request.batch.operations.length;

    if (success) {

        /* all operations in batch succeeded */
        /* for example, redisplay records in list */
        cdo.foreach( function(record) {
            /* reference the record/field as record.data.fieldName */
        });

    }
    else {
        /* one or more operations in batch failed */
        for (var idx = 0; idx < len; idx++) {

            var operationEntry = request.batch.operations[idx];


            switch (operationEntry.operation) {
                   case 1:
                       console.log("Operation: Create");
                       break;
                   case 3:
                       console.log("Operation: Update");
                       break;
                   case 3:
                       console.log("Operation: Delete");
                       break;
                   default:
                       console.log("Operation: Unexpected Code:" +
                                operationEntry.operation);
            }

            if (!operationEntry.success) {

              /* handle error condition */
              if (operationEntry.response && operationEntry.response._errors
 &&
                    operationEntry.response._errors.length > 0) {

                    var lenErrors = operationEntry.response._errors.length;
                    for (var idxError=0; idxError < lenErrors; idxError++) {

                        var errors =
                        operationEntry.response._errors[idxError];
                        var errorMsg = errors._errorMsg;
                        var errorNum = errors._errorNum;
                        /* handle error */

                    }
                }
            }
            else {
                 /* operation succeded */
            }
        }
```

Example

The following code fragment calls saveChanges(true) (with Submit) to apply all the corresponding record changes to the backend in a single request, using a returned Promise object to handle the results:

```
/* Some code that adds a record to CDO memory */
var newrec = mycdo.add( {State : 'MA'} );
/* Some code that updates a record in CDO memory */
myrecord = mycdo.find(function(myrec) {
        return (myrec.data.Name === 'Lift Tours');
});
myrecord.assign( {State: 'VT'} );
/* Some code that deletes a record from CDO memory */
myrecord = mycdo.find(function(myrec) {
    return (myrec.data.Name === 'Burrows Sport Shop');
});
myrecord.remove();

mycdo.autoApplyChanges = false;
mycdo.saveChanges(true).done( /* Successful Submit operation */
function( mycdo, success, request ) {
    /* All record changes processed by the Submit succeeded */
    /* Do additional processing… */
   mycdo.acceptChanges();

}).fail( /* Unsuccessful Submit operation */
    function( mycdo, success, request ) {
        /* check for OpenEdge errors on the Submit operation itself */
        console.log("Operation: Submit");
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {
            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {
                var error = request.response._errors[idxError];
                /* handle error results . . . */
                console.log("Error: " + error._errorNum + " " + error._errorMsg);
            }
        }
        /* Check for errors on each record change */
        var recErrors = mycdo.getErrors();
        if (recErrors.length > 0) {
        /* one or more Submitted record changes failed */
        /* get number of record changes sent */
        var len = request.jsrecords.length;
        for (var idx = 0; idx < len; idx++) {
            var myrecord = request.jsrecords[idx];
            var recError = myrecord.getErrorString();
            if (recError) {
                /* handle record-change error */
                console.log("Record change: " + myrecord.data["prods:rowState"]);
                console.log("Error: " + recError);
                myrecord.rejectChanges();
            }
            else {
                /* record change succeeded . . . */
                myrecord.acceptChanges();
            }
        }
    } /* end recErrors > 0 */
});
```

afterUpdate event

Fires after the CDO, by means of a `saveChanges( )` call following an `assign( )` or update() call, sends a request to update a record and receives a response to this request from the Cloud Data Server.

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

Syntax

```
function ( cdo , record , success , request )
```

`cdo`

A reference to the CDO that invoked the update operation. For more information, see the description of CDO property of the request object.

`record`

A reference to the table record upon which the update operation acted. For more information, see the description of record property of the request object.

`success`

A `Boolean` that is `true` if the update operation was successful. For more information, see the description of success property of the request object.

`request`

A reference to the request object returned after the update operation completes. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onAfterUpdate`, to handle the `afterUpdate` event fired on the CDO, `mycdo`:

```
/* subscribe to event */
mycdo.subscribe( 'afterUpdate', onAfterUpdate );

/* some code that would update a record and send to the server */
var record = mycdo.findById(myid);
record.assign( updatedDataObject );
mycdo.saveChanges();

function onAfterUpdate ( cdo , record , success , request ) {
    var jsrecError;

    if (success) {

        /* for example, get the values updated by the server from the record
           to redisplay */
        var newValue = record.data.myField;
        . . .
    }
    else {
        /* Check for errors on the Update operation */
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */
            }
        }
        /* Call getErrorString() on request.jsrecord to return any
         * record-change row error in before-image data (if present)
         * and handle the error */
        jsrecError = record.getErrorString();
        if (jsrecError) {
            /* process current record-change error */
        }

    }
};
```

assign( ) method (CDO class) (Same as update() method)

Updates field values for the specified `CloudDataRecord object` in CDO memory.

The specified record object can be either the working record of a CDO table reference or any record provided by a `CloudDataRecord object`.

After execution, any working records previously set before the method executed remain as the working records.To synchronize the change on the Cloud Data Server, call the `saveChanges( )` method.

Return type: `Boolean`

Syntax

```
record-ref.assign ( update-object )
cdo-ref.assign ( update-object )
cdo-ref.table-ref.assign ( update-object )
```

record-ref

A reference to a `CloudDataRecord object` for a table record in

CDO memory. You can obtain a `CloudDataRecord object` by:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`, `afterDelete`, or `afterDelete` event.

cdo-ref

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the CDO that has a working record.

update-object

Passes in the data to update the specified record object in CDO memory. Each property of the object has the name of a table field and the value to set for that field in the specified record. Any table fields without corresponding properties in *`update-object`* remain unchanged in the record.

Note: After this method updates the specified record object, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `String` fields are compared according to the value of the `caseSensitive` property.

Example

The following code fragment shows a jQuery event defined on a save button to save the current field values for a customer detail form to the corresponding `eCustomer` record in CDO memory:

```
dataSet = new progress.data.CDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.assign(update-object);
  dataSet.saveChanges();
});
```

The form has been displayed with previous values of the same record. When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`record`) and invokes the `assign( )` method on `record` with an object parameter to update the fields in `eCustomer` with any new values entered into the form.

async property

A `Boolean` that indicates, if set to `true`, that the Data Object operation was executed asynchronously on the Mobile or Web application.

Data type: `Boolean`

Access: Read-only

The `async` property is available only for the following CDO events:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (CDO`Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO. The object is also returned as the value of any CDO invocation method that you execute synchronously.

authenticationModel property

A string constant that specifies the type of authentication that the server requires from the Mobile or Web application.

Data type: `String`

Access: Readable/Writable

Valid values are:

- **`CDOSession.AUTH_TYPE_ANON`** — No authentication is required. This is the default value.

- CDO`Session.AUTH_TYPE_BASIC` — The Mobile or Web application requires a valid user ID and password, but does not provide a page containing a login form (credentials are typically entered in a generic login dialog provided by either the Mobile or Web application, the browser, or the native device container in which the App is running).The Mobile or Web application requires a valid user ID and password, but does not provide a page containing a login form (credentials are typically entered in a generic login dialog provided by either the Mobile or Web application, the browser, or the native device container in which the application is running).

- **`CDOSession.AUTH_TYPE_FORM`** — The Mobile or Web application requires a valid user ID and password and provides a page containing a login form.

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

autoApplyChanges property

A `Boolean` on a CDO that indicates if the CDO automatically accepts or rejects changes to CDO memory when you call the `saveChanges( )` method.

When set to `true`, and after you have invoked the `saveChanges( )` method, the CDO accepts all changes to CDO memory that are successfully applied on the Cloud Data Server, and rejects all changes from CDO memory that are completed with an error.

The default setting is `true` (which matches the behavior of previous releases). You can set this

property both during CDO instantiation and on an existing CDO.

Data type: `Boolean`

Access: Readable/Writable

When set to `false`, you must invoke one of the following methods at the appropriate time to accept or reject the changes in CDO memory:

- `acceptChanges( )`
- `acceptRowChanges( )`
- `rejectChanges( )`
- `rejectRowChanges( )`

You typically invoke one of these methods in the appropriate event handler for a CDO event associated with execution of the `saveChanges( )` method.

Example

The following code fragment sets the property both when the CDO is instantiated and after it is instantiated:

```
var cdoCustomers = new progress.data.CDO( { autoApplyChanges : false } );
. . .
cdoCustomers.autoApplyChanges = true;
```

autoSort property

A `Boolean` on a CDO and its table references that indicates if record objects are sorted automatically on the affected table references in CDO memory at the completion of a supported CDO operation.

When set to `true`, and after you have specified a sorting method for each affected table reference, record objects are sorted after the CDO operation completes its update of CDO memory. When set to `false`, or if no sorting method is specified for a given table reference, no automatic sorting occurs after the CDO operation completes. The default setting is `true` for all table references of a CDO.

Data type: `Boolean`

Access: Readable/Writable

When set on a CDO, the property setting affects the sorting of record objects for all table references in the CDO. When set on a single table reference, the property setting affects the sorting of record objects only for the specified table reference. For example, to set this property to `true` on only a single table reference in the CDO:

1. Set the value on the CDO to `false`, which sets `false` on all its table references.
2. Set the value on the selected table reference to `true`, which sets `true` on only the this one table reference.

In order to activate automatic sorting for an affected table reference, you must invoke one of the following CDO methods to specify a sorting method for the table reference:

- **`setSortFields( )`** — Identifies the *sort fields* to use in the record objects and whether each field is sorted in ascending or descending order according to its data type. Any `String` fields specified for a table reference are sorted using letter case according to the setting of the `caseSensitive` property (`false` by default).

Note: Changing the value of the `caseSensitive` property triggers an automatic sort if the

`autoSort` property is also set to `true` for the affected table reference.

- **`setSortFn( )`** — Identifies a *sort function* that compares two record objects according to the criteria you specify and returns a value that indicates if one record sorts later than the other in the sort order, or if the two records sort at the same position in the sort order. The `caseSensitive` property setting has no effect on the operation of the specified sort function unless you choose to involve the setting of this property in your criteria for comparison.

If you specify both sort fields and a sort function to sort the record objects for a table reference, the sort function takes precedence. You can also call the `setSortFields( )` and `setSortFn( )` functions to clear one or both settings of the sort fields and sort function. However, at least one setting must be active for automatic sorting to occur on a table reference.

The following supported CDO operations trigger automatic sorting on any affected table references before they complete their updates to CDO memory:

- **Invoking the `add( )` method** — Sorts the record objects of the affected table reference.
- **Invoking the `addRecords( )` method** — Sorts the record objects of either the single affected table reference or all affected table references in the CDO. (Unaffected table references do not participate in the sort, including those for which `autoSort` is `false`, those for which no sort fields or sort function are set, or those other than the single CDO table reference on which `addRecords( )` is called, if it is called only on a single table reference.)
- **Invoking the `assign( )` method (CDO class)** — Sorts the record objects of the affected table reference.
- **Assigning a value to a field reference directly on the working record of a table reference (*`cdo-ref`*`.`*`table-ref`*`.`*`field-ref`*` = `*`value`*)** — Sorts the record objects of the affected table reference.

Note: Assignment to a field referenced on the `data` property **never** triggers automatic sorting (for example, *`cdo-ref`*`.`*`table-ref`*`.data.`*`field-ref`*` = `*`value`*)

- **Changing the value of the `caseSensitive` property** — Sorts the record objects of the affected table reference, or of all affected table references if the property value is changed on the CDO.
- **Invoking either the `acceptRowChanges( )` or `rejectRowChanges( )` method** — Sorts the record objects of the affected table reference.
- **Invoking either the `acceptChanges( )` or `rejectChanges( )` method** — Sorts the record objects of all affected table references in the CDO. (Unaffected table references do not participate in the sort, including any table references for which `autoSort` is `false`, or for which no sort fields or sort function are set.)
- **Invoking the `fill( )` method** — Sorts the record objects of all affected table references in the CDO. (Unaffected table references do not participate in the sort, including any table references for which `autoSort` is `false`, or for which no sort fields or sort function are set.)

Note: Invoking the `remove( )` method does not trigger an automatic sort and has no effect on any existing sort order established for the table reference. However, if there is a sort order that depends on the presence or absence of the record object you are removing, and you want to establish the appropriate sort order when this record object is absent, you must manually sort the remaining record objects using the `sort( )` method by passing it the same sort function that you used to establish the sort order when this record object was present.

Caution: Because automatic sorting executes in JavaScript on the client side, sorting a large set of record objects can take a significant amount of time and make the UI appear to be locked. You might set a wait or progress indicator just prior to any action that can sort a large record set to alert the user that the app is working.

Example

In the following code fragment, automatic local sorting is turned off for all table references of the `dsCustOrds` CDO by setting its `autoSort` property to `false`. Automatic sorting is then turned on for the `eCustomer` table reference of the CDO by setting its `autoSort` value to `true` and using the `setSortFields( )` method to set its `Name` field as the single, descending sort field:

```
dsCustOrds = new progress.data.CDO( { name: 'dsCustomerOrders' });
dsCustOrds.autoSort = false.
dsCustOrds.eCustomer.autoSort = true.
dsCustOrds.eCustomer.setSortFields( "Name:DESC" );
dsCustOrds.fill();
. . .
```

When the `fill( )` method executes on the CDO, all the referenced tables are loaded from the Cloud Data Server into CDO memory with their record objects already sorted in case-insensitive, primary key order (by default). The record objects for `eCustomer` are then sorted locally in case-insensitive, descending order of the `Name` field.

batch property

A reference to an object with a property named `operations`, which is an array containing the request objects for each of the one or more Data Object record-change operations performed in response to calling the CDO `saveChanges( )` method either with an empty parameter list or with the single parameter value of `false`.

Data type: `Object`

Access: Read-only

The `batch` property is available only for the following CDO events, and **only** after calling

`saveChanges( )` with an empty parameter list, or with the single parameter value of `false`:

- `afterSaveChanges`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (`Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO.

beforeCreate event

Fires before the CDO, by means of a `saveChanges( )` call which makes an `add( )` or create call, sends a request the Cloud Data Server to create a record.

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

Syntax

```
function ( cdo , record , request )
```

cdo

A reference to the CDO that is invoking the create operation. For more information, see the description of cdo property of the request object.

record

A reference to the table record upon which the create operation is about to act. For more information, see the description of record property of the request object.

request

A reference to the request object returned before the create operation begins. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeCreate`, to handle the `beforeCreate` event fired on the CDO, `mycdo`, by assigning data to the newly created record before sending it to the server:

```
/* subscribe to event */
mycdo.subscribe( 'beforeCreate', onBeforeCreate );

/* some code that would add a record and save it */
var record = mycdo.add();

. . .

mycdo.saveChanges();

function onBeforeCreate( cdo , record , request ) {
    /* for instance,  here you can update data in the record
       before it is sent to the server */
    record.assign( { myField1 = myvalue, myField2 = myvalue2 } );
};
```

beforeDelete event

Fires before the CDO, by means of a `saveChanges( )` call which makes a a `remove( )` call, sends a request the Cloud Data Server to delete a record.

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

Syntax

```
function ( cdo , record , request )
```

cdo

A reference to the CDO that is invoking the delete operation. For more information, see the description of cdo property of the request object.

record

A reference to the table record upon which the delete operation is about to act. For more information, see the description of record property of the request object.

request

A reference to the request object returned before the delete operation begins. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeDelete`, to handle the `beforeDelete` event fired on the CDO, `mycdo`, where `myid` is the known ID of a record to find and delete:

```
/* subscribe to event */
mycdo.subscribe( 'beforeDelete', onBeforeDelete );

/* some code that would delete a record and send to the server */
var record = mycdo.findById( myid );
record.remove();
mycdo.saveChanges();

function onBeforeDelete( cdo , record , request ) {
    /* code to execute before sending request to the server */

};
```

beforeFill event

Fires before the CDO, by means of a `fill( )` call, sends a request to the Cloud Data Server to read a table or ProDataSet into CDO memory.

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

Syntax

```
function ( cdo , request )
```

`cdo`

A reference to the CDO that is invoking the read operation. For more information, see the description of cdo property of the request object.

`request`

A reference to the request object returned before the read operation begins. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeFill`, to handle the `beforeFill` event fired on the CDO, `mycdo`:

```
mycdo.subscribe( 'beforeFill', onBeforeFill );
mycdo.fill();

function onBeforeFill ( cdo , request ) {
    /* for instance, do any preparation to receive data from the server */
};
```

beforeInvoke event

Fires when a non-built-in method is called asynchronously on a CDO, before the request for the operation is sent to the Cloud Data Server.

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

Syntax

```
function ( cdo , request )
```

cdo

A reference to the CDO that is invoking the method. For more information, see the description of cdo property of the request object.

request

A reference to the request object returned before the operation begins. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeInvokeMyMethod`, to handle the `beforeInvoke` event fired on the CDO, `mycdo`, for an invocation of the `myMethod( )` invocation method passed the parameters specified by `paramObject`:

```
mycdo.subscribe( 'beforeInvoke', 'myMethod', onBeforeInvokeMyMethod );
mycdo.myMethod( paramObject );

function onBeforeInvokeMyMethod ( cdo , request ) {
    /* code to execute before sending request to the server */
};
```

beforeRead event

Fires before the CDO, by means of a `read( )` call, sends a request to the Cloud Data Server to read a table or ProDataSet into CDO memory.

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

Syntax

```
function ( cdo , request )
```

cdo

A reference to the CDO that is invoking the read operation. For more information, see the description of cdo property of the request object.

request

A reference to the request object returned before the read operation begins. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeRead`, to handle the `beforeRead` event fired on the CDO, `mycdo`:

```
mycdo.subscribe( 'beforeRead', onBeforeRead );
mycdo.fill();

function onBeforeRead ( cdo , request ) {
    /* for instance, do any preparation to receive data from the server */
};
```

beforeSaveChanges event

Fires once for each call to the `saveChanges( )` method on a CDO, before any create, update, or delete requests are sent to the Cloud Data Server.

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

Syntax

```
function ( cdo , request )
```

cdo

A reference to the CDO that is invoking the saveChanges method. For more information, see the description of cdo property of the request object.

request

A reference to the request object returned before the requested save operations begin. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeSaveChanges`, to handle the `beforeSaveChanges` event fired on the CDO, `mycdo`, where `myid` is the known ID of a record to find and process for Data Object operations being sent to the server:

```
mycdo.subscribe( 'beforeSaveChanges', onBeforeSaveChanges );

/* some code that would do multiple CUD operations and
   send them to the server */
var newrec = mycdo.add();
. . .
var record = mycdo.findById(myid);
record.remove(); mycdo.saveChanges();

function onBeforeSaveChanges ( cdo , request ) {
    /* code to execute before sending request to the server */
};
```

beforeUpdate event

Fires before the CDO, by means of a `saveChanges( )` call which makes an `assign( )` or update() call, sends a request the Cloud Data Server to update a record.

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

Syntax

```
function ( cdo , record , request )
```

cdo

A reference to the CDO that is invoking the update operation. For more information, see the description of cdo property of the request object.

record

A reference to the table record upon which the update operation is about to act. For more information, see the description of record property of the request object.

request

A reference to the request object returned before the update operation begins. For more information, see the description of request object.

Example

The following code fragment subscribes the function, `onBeforeUpdate`, to handle the `beforeUpdate` event fired on the CDO, `mycdo`, where `myid` is the known ID of a record to find and `updateDataObject` is an object containing the field values to assign in the found record. In this case, the onBeforeUpdate event handler assigns additional data to the updated record before sending it to the server:

```
/* subscribe to event */
mycdo.subscribe( 'beforeUpdate', onBeforeUpdate );

/* some code that would update a record and send to the server */
var record = mycdo.findById(myid);
record.assign( updateDataObject );

. . .

mycdo.saveChanges();

function onBeforeUpdate( cdo , record , request ) {
    /* for instance, here you can update data in the record
       further before it is sent to the server */
    record.assign( { myField1 = myvalue, myField2 = myvalue2 } );
};
```

caseSensitive property

A `Boolean` on a CDO and its table references that indicates if `String` field comparisons performed by supported CDO operations are case sensitive or case-insensitive for the affected table references in CDO memory.

When set to `true`, all supported comparisons on `String` fields for an affected table reference are case sensitive. When set to `false`, all supported comparisons on `String` fields for an affected table reference are case insensitive. The default setting is `false` for all table references of a CDO.

Data type: `Boolean`

Access: Readable/Writable

When set on a CDO, the property setting affects all table references in the CDO. When set on a single table reference, the property setting affects only the specified table reference. For example, to set this property to `true` on only a single table reference in the CDO:

1. Set the value on the CDO to `false`, which sets `false` on all its table references.
2. Set the value on the selected table reference to `true`, which sets `true` on only the one table reference.

The CDO operations that follow this property setting in `String` field comparisons include:

- Sorting record objects in CDO memory, including automatic sorting using sort fields that you specify using the `autoSort` property and the `setSortFields( )` method, and manual sorting using specified sort fields that you perform using the `sort( )` method

  Note: Changing the value of this property triggers an automatic sort if the `autoSort` property is also set to `true` for the affected table reference.

- Merging record objects into CDO memory for all merge modes that perform record field comparisons during execution of the `addRecords( )` method

Note: Any default `String` field comparisons that you might do in JavaScript within the callback functions that you specify for other CDO methods and events are always case sensitive according to JavaScript rules and ignore this property setting.

Note: To conform to Unicode default letter case mapping, the CDO support for case-insensitive `String`-field comparison and sorting relies on the `toUpperCase( )` JavaScript function instead of the `toLocaleUpperCase( )` JavaScript function. The latter function uses the locale letter case mapping, which might be different from the default letter case mapping in Unicode.

Example

In the following code fragment, automatic local sorting is set up for the `eCustomer` table reference on the `dsCustOrds` CDO, with its `Name` field as the single descending sort field. All other table references on `dsCustOrds` have no automatic local sorting set up by default. Because OpenEdge sorting on `String` fields is case-insensitive by default, the fragment makes the local sort on the `Name` field case sensitive by setting `caseSensitive` on `eCustomer` to `true`:

```
dsCustOrds = new progress.data.CDO( { name: 'dsCustomerOrders' } );
dsCustOrds.autoSort = false.
dsCustOrds.eCustomer.autoSort = true.
dsCustOrds.eCustomer.setSortFields( "Name:descending" );
dsCustOrds.eCustomer.caseSensitive = true.dsCustOrds.fill();
. . .
```

When the `fill( )` method executes on the CDO, after all the referenced tables are loaded from the Cloud Data Server, with their record objects already sorted in case-insensitive, primary key order (by default), the record objects for `eCustomer` are then sorted locally in case-sensitive, descending order of the `Name` field.

catalogURIs property

Returns the list of URIs used to load the CDO catalogs to access the Cloud Data Services provided by the Mobile or Web application for which the current **CDO**`Session` object manages a user login session.

Data type: `String` array

Access: Read-only

This list includes the URI for each CDO catalog loaded using the `addCatalog( )` method. To return a corresponding list of Cloud Data Service names for which the CDO catalogs are loaded, read the `serviceNames` property.

cdo property

An object reference to the CDO that performed the operation returning the request object.

Data type: CDO class

Access: Read-only

The `cdo` property is available for all CDO events. This request object property is also available for any session `online` and `offline` events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (`Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO. The object is also returned as the value of any CDO invocation method that you execute synchronously.

CDOs property

Returns an array of CDOs that use the current **CDO**`Session` object to communicate with their Data Object services.

Data type: CDO

array **Access:**

Read-only

clientContextId property

The value of the most recent client context identifier (CCID) that the **CDO**`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

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

Note: You can access this `OERequestInfo` object on the Cloud Data Server using the

`CURRENT-REQUEST-INFO` attribute of the Cloud Data Server `SESSION` system handle. This CCID value is also available as the `SESSION-ID` attribute of the single sign-on (SSO) client-principal handle returned by the `GetClientPrincipal( )` method of the same `OERequestInfo` class-based object.

connected property

Returns a `Boolean` that indicates the most recent online status of the current **CDO**`Session` object, when it last determined if the Mobile or 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 or 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 or 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

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

- A successful result of the `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 `Session` object executing its `logout( )` method, which sets the property to `false`.
- The `Session` object receiving an `offline` or `online` event from its window object.
- The result of the `Session` object executing its `ping( )` method.

create() method (Same as add() method)

Introduced: Mobile Release 4.1

Syntax

```
jsdo=ref.create ( [new-record-object] )
jsdo=ref.table-ref.create ( [new-record-object] )
```

Example

```
var dataSet = new Progress.data.JSDO( 'CustomerOrderDS' );
dataSet.customer.create( { CustNum: 1000, Balance: 10000, State: 'MA' } );

// CustNum is set automatically by using the relationship
dataSet.order.create( { OrderNum: 1000 } );
```

data property

The data (field values) for a record associated with a `CloudDataRecord object`.

Data type: `Object`

Access: Read-only

The returned object contains a field reference property (*`field-ref`* in syntax) for each field (column) in the table, where the property name is identical to a table field name and the property value for the corresponding JavaScript data type.

You can obtain a `CloudDataRecord object` by invoking one of the CDO methods that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`) or by accessing the `record` property on a CDO table reference that already has a working record.

Note: If a given CDO table has a working record, you can access each *`field-ref`* of the working record directly on the corresponding table reference property (*`table-ref`*) of the CDO. For the working record of a table reference, then, references to the `CloudDataRecord object` of the working record and its `data` property are both implied by the table reference alone.

Caution: Never write directly to a *`field-ref`* using this `data` property; in this case, use *`field-ref`* **only to read** the data. Writing field values using the `data` property does **not** mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in CDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a *`cdo-ref.table-ref.field-ref`* for a working record or by calling the `assign( )` method on a valid *`table-ref`* or `CloudDataRecord object` reference. For information on table references (*`table-ref`*), see the reference entry for the table reference property (CDO).

deleteLocal( ) method

Clears out all data and changes stored in a specified local storage area, and removes the cleared storage area.

Return type: `undefined`

Syntax

```
deleteLocal ( [ storage-name ] )
```

`storage-name`

The name of the local storage area to be removed. If `storage-name` is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `cdo_serviceName_resourceName`, where *`serviceName`* is the name of the Cloud Data Service that supports the CDO instance, and *`resourceName`* is the name of the resource (table, dataset, etc.) for which the CDO instance is created.

If this method encounters any errors, it leaves the specified storage area unchanged and throws an exception.

Example

The following code fragment clears out all the data currently stored in the default storage area and removes the storage area:

```
dataSet = new progress.data.CDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
.
.
.
dataSet.deleteLocal();
```

fill( ) method (Same as read() method)

Initializes CDO memory with record objects from the data records in a single-table resource, or in one or more tables of a multi-table resource, as returned by the read operation of the Data Object resource for which the CDO is created.

This method always executes asynchronously and returns results (either or both) in subscribed CDO event callbacks or in callbacks that you register using methods of a concurrency object such as a promise object in JavaScript. The fill() method returns a concurrency object if it is available to the system.

After completing execution, the working record for each referenced table is set to its first record, depending on any active parent-child relationships (for a multi-table resource) and automatic sort settings. So, for each child table, the first record object is determined by its table reference sort order (if any) and its relationship to the related working record in its parent table.

Syntax

```
fill ( [   parameter-object   |   filter-string ] )
```

parameter-object

An Object initialized with a set of properties that can be used on the server to select, sort, and page the records to be returned. If you are using the JSDO dialect of the Kendo UI DataSource to bind a JSDO instance to Kendo UI widgets, the DataSource initializes and invokes fill( ) with parameter-object. The properties you can initialize when you call fill( ) directly include:

- **filter** — An object containing property settings used to select the records to be returned. These property settings are in the format as the property settings in the Kendo UI DataSouce filter property object. For more information, see the filter configuration property description in the Telerik Kendo UI DataSource documentation.
- **id** — A string specifying a unique ID that the resource understands to identify a specific record.
- **skip** — A number that specifies how many records to skip before returning (up to) a page of data. You must specify this property together with the top property**.**
- **sort** — An expression that specifies how to sort the records to be returned.
- **tableRef** — A string specifying the name of a table reference in the CDO. This property is required when the CDO represents a multi-table resource and the filter property Object is **also** specified with filter information.
- **top** — A number that specifies how many records (the page size) to return in a single page of data after using skip. You must specify this property together with the skip property. The final page of a larger result set can contain a smaller number of records than top specifies.

Note: You must specify both skip and top to implement server paging. If these properties are not specified together, the resource's read operation determines the records to return without using any paging information.

filter-string

A string that can be used on the Cloud Data Server to select records to be returned, much like the `WHERE` option of the record phrase. The actual format of this string and its affect on the records returned is determined by the routine on the Cloud Data Server that uses it. For example, you might pass:

- A single key value (e.g., `"30"`)
- A relational expression (e.g., `"CustNum > 10 AND CustNum < 30"`)
- An actual `WHERE` string (e.g., `'Item.CatDescription CONTAINS "ski & (gog* ! pol*)"'`)
- A string using the JSON Filter Pattern

Note: The CDO requires the URI for the `"read"` operation of the resource to contain the following query string: `"?`*`filter`*`=~{`*`filter`*`~}"`, where *`filter`* is the name of a string input parameter defined for the routine that implements the operation (`INPUT` *`filter`* `AS CHARACTER`).

Caution: Using an actual `WHERE` string for a dynamic query can create a potential security issue.

If you do not specify *`filter-string`*, the records returned, again, depend on the routine.

Example

The following code fragment shows the fill( ) method invoked on a JSDO for an OpenEdge single-table resource (dsCustomer), with results returned using the afterFill event:

```
dataSet = new progress.data.JSDO( 'dsCustomer' );
dataSet.subscribe("afterFill", onAfterFill);

dataSet.fill();

function onAfterFill( jsdo , success , request ) {
   if (success) {

      /* for example, add code to display all records on a list */
      jsdo.foreach(function (jsrecord) {
         /* you can reference the fields as jsrecord.data.field */
      });
   }
   else {
    if (request.response && request.response._errors &&
      request.response._errors.length > 0) {

      var lenErrors = request.response._errors.length;
      for (var idxError=0; idxError < lenErrors; idxError++) {

        var errorEntry = request.response._errors[idxError];
        var errorMsg = errorEntry._errorMsg;
        var errorNum = errorEntry._errorNum;
        /* handle error */

      }
    }
  }

};
```

The following code fragment shows the fill( ) method invoked on a JSDO for a similar OpenEdge single-table resource (dsCustomerOrder), with results returned using a Promise object:

```
dataSet = new progress.data.JSDO( 'dsCustomer' );

dataSet.fill().done(
  function( jsdo, success, request ) {
    /* for example, add code to display all records on a list */
    jsdo.foreach(function (jsrecord) {
      /* you can reference the fields as jsrecord.data.field */
    });
  }).fail(
  function( jsdo, success, request ) {
    if (request.response && request.response._errors &&
      request.response._errors.length > 0) {

      var lenErrors = request.response._errors.length;
      for (var idxError=0; idxError < lenErrors; idxError++) {

        var errorEntry = request.response._errors[idxError];
        var errorMsg = errorEntry._errorMsg;
        var errorNum = errorEntry._errorNum;
        /* handle error */
      }
    }
  });
```

Using a Promise object, the Promise done and fail functions do not have to test the success parameter for a successful (true) or failed (false) execution of the fill( ) method, because done executes only when fill( ) succeeds and fail executes only when fill( ) fails.

find( ) method

Searches for a record in a table referenced in CDO memory and returns a reference to that record if found. If no record is found, it returns `null`.

After completing execution, any record found becomes the working record for the associated table. If the searched table has child tables, and the `useRelationships` property is `true`, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent. If a record is not found, the working record is not set, and the working records of any child tables are also not set.

Return type: CloudDataRecord object

Syntax

```
cdo-ref.find ( funcRef )
cdo-ref.table-ref.find ( funcRef )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`funcRef`

A reference to a JavaScript callback function that returns a `Boolean` value and has the following signature:

Syntax:

```
function [ func-name ] ( record-ref )
```

Where *func-name* is the name of a callback function that you define external to the `find( )` parameter list and *record-ref* is a `CloudDataRecord` reference to the next available record on the specified table reference. You can then pass *func-name* to the `find( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

The `find( )` method executes your *funcRef* callback for each record of the table reference, until it returns `true`, indicating that the callback has found the record. You can then test the field values on the `data` property of *record-ref* to determine the result. Otherwise, your callback returns `false` and the `find( )` method executes the callback again on the next available record.

If your *funcRef* callback finds the record, `find( )` completes execution with both its return value and the `record` property of the associated table reference set to the `CloudDataRecord` reference of the found working record. If `find( )` reaches the end of the available records without *funcRef* returning `true`, `find( )` completes execution with both its return value and the `record` property on the table reference set to `null`, indicating that the sought for record was not found.

If the associated table reference is for a child table in a ProDataSet, if the `useRelationships` property is `true`, `find( )` uses the relationship to filter out all but the child records of the working record in the parent table. However, if the working record of the parent is not set, `find( )` throws an error. If `useRelationships` is `false`, the search includes all records of the child table and no error is thrown.

Example

In following code fragment, `cdo` references a single `customer` table:

```
var cdo = new progress.data.CDO( 'customer' );

cdo.find(function(record) {
  return (record.data.CustNum == 10);
});
```

The inline function passed to `find( )` returns `true` or `false` based on the value of the `CustNum` property of the object returned by the `data` property for the currently available `CloudDataRecord` reference.

findById( ) method

Locates and returns the record in CDO memory with the internal ID you specify.

If no record is found, it returns `null`. You can access the internal ID of the working record of a table reference, or any `CloudDataRecord object`, by calling the `getId( )` method.

After completing execution, any record found becomes the working record for the associated table. If the searched table has child tables, and the `useRelationships` property is `true`, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent. If a record is not found, the working record is not set, and the working records of any child tables are also not set.

Return type: CloudDataRecord object

Syntax

```
cdo-ref.findById ( id )
cdo-ref.table-ref.findById ( id )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`id`

The internal record ID used to match a record of the table reference. This is the same value originally returned for the record using the `getId( )` function. It is typically used to create a jQuery listview row to display the record or a detail form used to display the record in the current HTML document. Later, when a listview row or detail form is selected, the corresponding `id` attribute with this value can be used to return the record from the CDO, possibly to update the record with new data values input by the user.

If `findById( )` locates a record with the matching record ID, it completes execution with both its return value and the `record` property of the table reference set to the `CloudDataRecord` reference of the found working record. If the function does not locate the record, it completes execution with both its return value and the `record` property on the table reference set to `null`, indicating that no record of the table reference has a matching internal record ID.

If the table reference references a child table in a ProDataSet, when the `useRelationships` property is `true`, `findById( )` uses the relationship to filter out all but the child records of the working record in the parent table; the remaining child records are excluded from the search. If `useRelationships` is `false` or the working record of the parent is not set, the search includes all records of the child table and no error is thrown.

Example

The following code fragment shows a jQuery event defined on a save button to save the current field values for a customer detail form to the corresponding `eCustomer` record in CDO memory:

```
dataSet = new progress.data.CDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.assign();
  dataSet.saveChanges();
});
```

The form has been displayed with previous values of the same record. When the button is clicked, the event handler finds the original `eCustomer` record by calling `findById( )` with the `id` attribute of the form (`$('#custdetail #id').val()`), which is set to the internal ID of the record. The `record.assign( )` method then updates the record from the values of the corresponding form fields and `saveChanges( )` invokes the resource "`update`" operation on the Cloud Data Server to save the updated record to its data source.

fnName property

For an invoke operation, the name of the CDO invocation method that called the operation.

The `fnName` property is null in the case of a request object returned by a built-in create, read, update, or delete operation.

Data type: `String`

Access: Read-only

Applies to: request object

The `fnName` property is available only for the following CDO event:

- `afterInvoke`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (CDO`Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO. The object is also returned as the value of any CDO invocation method that you execute synchronously.

Note: The value of the `fnName` property is the same as that of the *`op-name`* parameter passed to the `subscribe( )` method that subscribed to the current invoke operation event.

foreach( ) method

Loops through the records of a table referenced in CDO memory and invokes a user-defined callback function as a parameter on each iteration.

With each iteration, it also sets the current record as the working record and passes it as a parameter to the callback function. This function can then operate on the working record and return a value indicating whether the `foreach( )` terminates the loop or invokes the callback function on the next working record of the table.

If the referenced table has child tables, and the `useRelationships` property is `true`, with each iteration through the loop, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent.

After completing execution, the working records of the associated table, and any child tables, are the most recent working records established when the method terminates the loop.

Return type: `null`

Syntax

```
cdo-ref.foreach ( funcRef )
cdo-ref.table-ref.foreach ( funcRef )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`funcRef`

A reference to a JavaScript callback function that returns a `Boolean` value and has the following signature:

Syntax:

```
function [ func-name ] ( record-ref )
```

Where *`func-name`* is the name of a callback function that you define external to the `foreach( )` parameter list and *`record-ref`* is a `CloudDataRecord object` reference to the next working record on the table reference. You can then pass *`func-name`* to the `foreach( )` method as the *`funcRef`* parameter. Alternatively, you can specify *`funcRef`* as the entire inline function definition without *`func-name`*.

The `foreach( )` method executes your *`funcRef`* callback for each record of the table reference, making this record the working record and passing it in as *`record-ref`*. You can then access the field values of the working record using the `data` property on *`record-ref`* or any field references available from the table reference. You can also invoke other CDO methods, for example, to operate on the working record, including additional calls to `foreach( )` to operate on working records of any child tables.

Your *`funcRef`* callback can terminate the `foreach( )` loop by returning `false`. If the callback does not return `false`, the loop continues.

If the table reference references a child table in a ProDataSet, when the `useRelationships` property is `true`, `foreach( )` uses the relationship to filter out all but the child records of the working record in the parent table. However, if the working record of the parent is not set, `foreach( )` throws an error. If `useRelationships` is `false`, the loop includes all records of the child table and no error is thrown.

Example

After creating a CDO for a `dsCustomer` resource and loading it with record objects, the following code fragment shows the `foreach( )` method looping through `eCustomer` records in CDO memory and displaying the `CustNum` and `Name` fields from each record, one record per line, to the current HTML page, and also to the console log:

```
cdo = new progress.data.CDO({ name: 'dsCustomer' });
cdo.subscribe( 'AfterFill', onAfterFillCustomers, this );

cdo.fill();

function onAfterFillCustomers(cdo, success, request) {
  cdo.eCustomer.foreach( function(customer) {
    document.write(customer.data.CustNum + ' ' + customer.data.Name + '<br>');
    console.log(customer.data.CustNum + ' ' + customer.data.Name);
  } );
};
```

getData( ) method

Returns an array of record objects for a table referenced in CDO memory.

If this is a child table, and the `useRelationships` property is `true`, the specific record objects in the result set depends on the relationship to its parent.

After completing execution, any working records previously set before the method executed remain as the working records.

Return type: `Object` array

Syntax

The getData() method supports 2 signatures:

Signature 1:

```
cdo-ref.getData ( )
cdo-ref.table-ref.getData ( )
```

Signature 2:

```
cdo-ref.getData ([ params ] )
cdo-ref.table-ref.getData ([ params ] )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`params`

Used when performing server side paging to specify top, skip, and sort.

getErrors() method [4.2]

Returns an array of errors from the most recent CDO operation.

Return type: array

Syntax

```
cdo.getErrors()
cdo.table-ref.getErrors()
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

Example using Node.js

```
XMLHttpRequest = require("./xhr/XMLHttpRequest").XMLHttpRequest;
var progressjs = require("./progress/progress.jsdo.js");

var session,
   jsdo,
   serviceURI = "http://oemobiledemo.progress.com/CustomerService",
   catalogURI =
"http://oemobiledemo.progress.com/CustomerService/static/mobile/CustomerService.json";

session = new progress.data.Session();
session.login(serviceURI, "", "");
session.addCatalog(catalogURI);

jsdo = new progress.data.JSDO({ name: 'Customer' });
jsdo.subscribe('AfterFill', onAfterFillCustomers, this);
jsdo.subscribe('AfterSaveChanges', onAfterSaveChanges, this);
jsdo.fill({top: 10, skip: 2});

function onAfterFillCustomers(jsdo, success, request) {
   jsdo.eCustomer.foreach(function(customer) {
      console.log(jsdo.eCustomer.Name);
   });

   var jsrecord = jsdo.find(function(){ return true; });
   jsrecord.assign({State: "ERROR"});
   jsdo.saveChanges();

}

function onAfterSaveChanges(jsdo, success, request) {
   var errors = jsdo.getErrors();
   console.log("DEBUG: AfterSaveChanges: " + success + " errors: " + errors.length);

   if (!success) {
      for (var i = 0; i < errors.length; i++) {
                                   console.log("DEBUG: Record id: " + errors[i].id);
                         console.log("ERROR: " + errors[i].error);
                      }
   }
}
```

getErrorString( ) method

Returns any before-image error string in the data of a record object referenced in CDO memory that was set as the result of a Data Object create, update, delete, or submit operation.

If there is no error string in the data of a record object, this method returns `undefined`.

The specified record object can be either the working record of a referenced table, or any record provided by a `CloudDataRecord object`.

After execution, any working records previously set before the method executed remain as the working records.

Return type: `String`

Syntax

```
record-ref.getErrorString ( )
cdo-ref.getErrorString ( )
cdo-ref.table-ref.getErrorString ( )
```

`record-ref`

A reference to a `CloudDataRecord object` for a table record in CDO memory. You can obtain a `CloudDataRecord object` by:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `records` property on the request object returned to the callback of a CDO `afterSaveChanges` event on completion of a Data Object submit operation.

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the CDO that has a working record.

The error string returned by this function contains error information associated with a temp-table buffer on the Cloud Data Server that results from a change to the record's before-image data. This error information is only returned for errors involving record updates associated with a ProDataSet that has before-image data and it can be returned for a Data Object create, update, delete, or submit operation (`saveChanges(true)`) that otherwise completes successfully.

getId( ) method

Returns the unique internal ID for the record object referenced in CDO memory.

The specified record object can be either the working record for a referenced table, or any record provided by a `CloudDataRecord object`.

After execution, any working records previously set before the method executed remain as the working records.

Return type: `String`

Syntax

```
record-ref.getId ( )
cdo-ref.getId ( )
cdo-ref.table-ref.getId ( )
```

`record-ref`

A reference to a `CloudDataRecord object` for a table

record in CDO memory. You can obtain a

`CloudDataRecord object` by:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already

has a working record.

- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`,
  `afterDelete`, or `afterDelete` event.

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the CDO that has a working record.

The internal record ID returned by this function is a unique value generated by OpenEdge for each record object loaded in CDO memory using the `fill( )`, `add( )`, or `addRecords( )` methods. To return and set the specified record as the working record, you can pass any value returned by this method to the `findById( )` method called on the associated table reference.

Note: The value returned by getId() for any given record object can change with each invocation of the `fill( )` method.

getSchema( ) method

Returns an array of objects, one for each field that defines the schema of a table referenced in CDO memory.

The properties of each object define the schema elements of the respective field.

After completing execution, any working records previously set before the method executed remain as the working records.

Return type: `Object` array

Syntax

```
cdo-ref.getSchema ( )
cdo-ref.table-ref.getSchema ( )
```

`cdo-ref`

A reference to the CDO.You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

hasData( ) method

Returns `true` if record objects can be found in any of the tables referenced in CDO memory (with or without pending changes), or in only the single table referenced on the CDO, depending on how the method is called; and returns `false` if no record objects are found in either case.

Return type: `Boolean`

Syntax

```
cdo-ref.hasData ( )
cdo-ref.table-ref.hasData ( )
```

`cdo-ref`

A reference to the CDO. If you call the method on *`cdo-ref`*, the method verifies if any data is available in the CDO, whether it is created for a single table or a ProDataSet with multiple tables.

`table-ref`

A table reference on the CDO. If you call the method on *`table-ref`*, the method verifies if any data is available in the referenced table. If the `useRelationships` property is `true`, this includes related data in any other CDO table(s) with which the referenced table has a parent-child relationship.

This method always returns `true` immediately after the `fill( )` method successfully loads CDO memory with one or more record objects.

Three cases where this method returns `false` are as follows:

1. After a CDO is instantiated but before `fill( )` or any other method (such as `addRecords( )`) has been invoked to load its CDO memory with records
2. After the `fill( )` method completes successfully, but returns no records because none match the specification of its *`filter`* parameter
3. After the `saveChanges( )` method completes successfully on a CDO, where its `autoApplyChanges` property set to `true` and all the records in the specified CDO, or its table reference, are marked for deletion

Two typical uses of this method include determining if there is any data in CDO memory that you might want to save in local storage using the CDO `saveLocal( )` method, or that you might not want to lose by replacing CDO memory, using the CDO `readLocal( )` method, with other data previously saved in local storage.

Example

The following code fragment shows an example of how you might use `hasData( )` to decide when to save CDO memory to local storage. It first invokes the `fill( )` method on a CDO (`dataSet`) to load CDO memory, and after a certain amount of work is done with the CDO, decides to save all the data in CDO memory to the default local storage area when it finds that records exist in CDO memory to save:

```
dataSet = new progress.data.CDO( 'dsStaticData' );
dataSet.fill();

/* Work done with the dataSet CDO memory */
.
.
.
if (dataSet.hasData())
{
  dataSet.saveLocal();
}
```

hasChanges( ) method

Returns `true` if CDO memory contains any pending changes (with or without before-image data), and returns `false` if CDO memory has no pending changes.

Return type: `Boolean`

Syntax

```
hasChanges ( )
```

This method always returns `true` if any change to CDO memory has marked a record object it contains as created, updated, or deleted.

This method always returns `false` if you execute it immediately after invoking any one of the following methods on the CDO:

- `fill( )`
- `saveChanges( )`, if the `autoApplyChanges` property is also set to `true`
- `acceptChanges( )`
- `rejectChanges( )`

A typical use of this method is to determine if there are any changes in CDO memory that you want to save to local storage using the CDO `saveLocal( )` method.

Example

The following code fragment shows an example of how you might use `hasChanges( )` to decide how to save CDO memory to local storage. It first invokes the `fill( )` method on a CDO (`dataSet`) to load CDO memory, sets the `autoApplyChanges` property on the CDO to not automatically accept or reject changes saved to the server based on the success or failure of the save, and after a certain amount of work is done with the CDO, decides to save **all** the data in CDO memory to the default local storage area, or to save **only** the pending changes, based on whether any pending changes currently exist in CDO memory:

```
dataSet = new progress.data.CDO( 'dsStaticData' );
dataSet.fill();
dataSet.autoApplyChanges = false;

/* Work done with the dataSet CDO memory */
.
.
.
if (dataSet.hasChanges())
{
  dataSet.saveLocal(progress.data.CDO.CHANGES_ONLY);
}
else
{
  dataSet.saveLocal(progress.data.CDO.ALL_DATA);
}
```

invocation method

Any method on the CDO that is defined by the resource to execute a corresponding `routine` on the Cloud Data Server as an invoke operation.

This can be any routine in the Data Object interface that is annotated with an "`invoke`" operation type. The invocation method name can be the same as the routine or an alias, as defined by the resource. The method passes any input parameters as properties of an object parameter. The method returns results from the routine, including any return value and output parameters, as properties of a request object that is returned by the method.

Note: The results of an invoke operation have no effect on CDO memory.

After completing execution, any working records previously set before the method executed remain as the working records.

Return type: request object

Syntax

```
op-name ( [ input-object [ , async-flag ] ] )
```

`op-name`

The name (specified as an identifier) of the invocation method as defined by the resource.

`input-object`

An object whose properties and values match the case-sensitive names and data types of the input parameters specified for the routine. If the routine does not take input parameters, specify `null` or leave out the argument entirely.

`async-flag`

A `Boolean` that when `true` causes the method to execute asynchronously and when

`false` causes the method to execute synchronously. The default value is `true`.

For a synchronous invocation, the method returns a request object that contains several properties depending on the status of the invoke operation. However, if there are any output parameters or a return value, they are returned as properties of an object referenced by the `response` property of the request object. The `response` object properties for output parameters match the

Case-sensitive names and data types of the output parameters specified for the routine. Any return type is returned by an OpenEdge-defined `_retVal` property with a matching data type.

For an asynchronous invocation, the method returns a similar request object as input to any event handler function subscribed to the following named events that fire in the following operational order:

1. beforeInvoke event
2. afterInvoke event

Note: If you are calling an invocation method that either sends a table or ProDataSet object as a property of *`input-object`* or returns a table or ProDataSet object as a property of the `response` property object, you need to apply a rule in order to access this table or ProDataSet object. The rule is that wherever you de-reference or reference a table or ProDataSet object, you must reference that value twice, separated by a period or a colon, depending on the context. The reason is that the table or ProDataSet name is both the name of the parameter defined for the routine and also the name of the JavaScript object containing the JSON data returned from the server. For example, to access a table object, `ttCust` returned by the `response` property in a `request` object, you must code the following de-reference: `request.response.ttCust.ttCust`. Similarly, if you pass `ttCust` to an invocation method, `InputTT( )`, you must code the following reference: `cdo.InputTT( {ttCust: {ttCust:ttCust}});`

Note: If the invocation method passes a ProDataSet as an input or output parameter, that ProDataSet can contain before-image data. However, the method does no processing of the before-image data in any way. You must therefore manage the object appropriately. For example, you can use an output ProDataSet containing before-image data as a *`merge-object`* parameter to the `addRecords( )` method as long as your CDO uses the same ProDataSet schema and the resource supports before-imaging.

invoke( ) method

Asynchronously calls a custom invocation method on the JSDO to execute an Invoke operation defined by a Data Object resource.

The asynchronous execution of the specified invocation method using invoke( ) returns results using a concurrency object such as a jQuery Promise in JavaScript. You can also directly call an invocation method on the CDO either synchronously, returning results as its return value, or asynchronously, returning results using a callback subscribed to the afterInvoke event.

For more information on the possible implementations for custom invocation methods, and how to call them directly on the CDO, see the description of the **invocation method**.

This method has no affect on existing working record settings.

Note: The results of an Invoke operation have no effect on CDO memory. However, you can use the CDO addRecords( ) method to merge any record data that is returned by an invocation method into CDO memory.

Syntax

```
invoke ( op-name [ , input-object ] )
```

op-name

A string that specifies the name of the invocation method as defined by the resource.

input-object

An object whose properties and values match the case-sensitive names and data types of the input parameters specified for the server routine that implements the invocation method. If the implementing routine does not take input parameters, specify null or leave out the argument entirely.

login method

This method logs in to a Web application, handling authentication according to the authentication model specified when the constructor was called. Will throw an error if it is not valid to send a request to the Web application (for example, if the CDOSession has already logged in successfully).

Return type: `undefined or a Concurrency object`

Syntax

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

userName
Ignored if using Anonymous authentication, necessary otherwise.

password
Ignored if using Anonymous authentication, necessary otherwise.

options
An object that has the following property:

iOSBasicAuthTimeout (optional)

This setting applies only if the login is an asynchronous request using Basic authentication made from an iOS device. The value is the time, in milliseconds, that the login() method will wait for a response before generating an error (an error may mean that the user entered invalid credentials). If this value is 0, no timeout will be set. If the iOSBasicAuthTimeout is not present, login() will use the default of 4 seconds.

(This property, and its implementation, is a workaround for a bug in Apache Cordova that may affect asynchronous requests sent from iOS devices using HTTP Basic authentication where the credentials are incorrect. Such requests have been found to hang rather than correctly call the response handler with a 401 Unauthorized status code.)

The login() method executes asynchronously. It may be implemented to communicate the result by firing an event named *afterLogin* or through the use of a concurrency object, for example a promise object in JavaScript. The signatures for *afterLogin* or for handlers associated with a concurrency object are the same:

session

Reference to the CDOSession object that login() was called on

result

Data type: Number

Indicates outcome of login:

CDOSession.AUTHENTICATION_SUCCESS
the login succeeded

CDOSession.AUTHENTICATION_FAILURE
authentication error on the login attempt

CDOSession.GENERAL_FAILURE
other error on the login attempt

info

a JavaScript object that can have the following properties:

errorObject

A reference to any Error object thrown during the login

xhr

Reference to the XMLHttpRequest object used to make the login request to the Web application

It's possible to log out from a Web application and then log in again using the same CDOSession object. The login will use the same serviceURI and authenticationModel originally passed to the constructor, but new credentials must be passed each time that login() is called.

loginHttpStatus property

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

Data type: `Number`
Access: Read-only

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 `CDOSession` object.

Data type: `Number`

Access: Read-only

Possible login return values include the following numeric constant values:

- **`CDOSession.LOGIN_SUCCESS`** — User login session started successfully.
- **`CDOSession.LOGIN_AUTHENTICATION_FAILURE`** — User login failed because of invalid user credentials.
- **`CDOSession.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.

logout method

Logs out from a session. Catalog information is retained, so CDOs that use catalogs loaded by the CDOSession object will still have some functionality, but they will not be able to make server requests, even if the Anonymous authentication model is being used. May throw an error.

Return type: `undefined or a Concurrency object`

Syntax

```
logout ( )
```

The logout() method executes asynchronously. It may be implemented to communicate the result by firing an event named *afterLogout* or through the use of a concurrency object, for example a promise object in JavaScript. The signatures for *afterLogout* or for handlers associated with a concurrency object are the same:

session
Reference to the CDOSession object that called logout()

result
Indicates outcome of logout:
CDOSession. SUCCESS the logout succeeded
CDOSession.GENERAL_FAILURE some error on the logout attempt

info
a JavaScript object that can have the following properties:

errorObject	A reference to any Error object thrown during the logout
xhr	Reference to the XMLHttpRequest object used to make the logout request to the Web application

offline event

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

Handler signature:

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

session

Reference to the CDOSession object that fired the event

offlineReason

A string that indicates the reason that the Web application is considered to be offline. Possible values are the following string constants (defined by progress.data.Session) :

```
CDOSession.DEVICE_OFFLINE        "Device is offline"
CDOSession.SERVER_OFFLINE        "Cannot contact server"
CDOSession.WEB_APPLICATION_OFFLINE
        "Mobile Web Application is not available"
progress.data.Session.APPSERVER_OFFLINE     "AppServer is not available"
```

request

If the offline condition was detected as a result of a request sent on behalf of a CDO, this is a reference to the request object used to make the request.

online event

Fires when the CDOSession object detects that the device on which it is running has become online after having been offline, or that the Mobile or Web application to which it has been connected is now available after it had been unavailable.

Handler signature:

function (session , request)

session

Reference to the CDOSession object that fired the event

request

If the offline condition was detected as a result of a request sent on behalf of a CDO, this is a reference to the request object used to make the request.

ping method

Determines whether the Mobile or Web application is available. In the case of an OpenEdge Web application, also determines whether its associated AppServer is running. Will cause an offline or online event to fire if the ping detects that there has been a change in online state. Will throw an error if it the CDOSession object has not logged in or has logged out.

Return type: `undefined or a Concurrency object`

Syntax

```
ping ( [ options ] )
```

options
An object that has the following property:
onCompleteFn (optional)
A callback function. It is called after the ping() method receives a response from the server or times out, regardless of the online status returned.

The ping() method executes asynchronously. It may be implemented to communicate the result via a callback function or through the use of a concurrency object, for example a Promise object in JavaScript. If implemented to use a callback function, the callback should be passed as a property named *onCompleteFn* in an object passed as an argument to ping(). The signatures for *onCompleteFn(),* or for handlers associated with a concurrency object, are the same:

session	Reference to the CDOSession object that called ping()
result	A boolean, *true* if the Web application is online, *false* if not

info
a JavaScript object that can have the following properties:
xhr
Reference to the XMLHttpRequest object used to make the ping request to the Web application

offlineReason
A string that indicates the reason that the Web application is considered to be offline. Possible values are the following string constants (defined by progress.data.Session) :

CDOSession.DEVICE_OFFLINE	"Device is offline"
CDOSession.SERVER_OFFLINE	"Cannot contact server"
CDOSession.WEB_APPLICATION_OFFLINE	"Mobile Web Application is not available"
CDOSession.APPSERVER_OFFLINE	"AppServer is not available"

pingInterval property

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

Setting this property to a value greater than zero (0) causes the `CDOSession` 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

You can set `pingInterval` to start the automatic execution of `ping( )` any time after you create the CDO`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 use a callback mechanism to get the results. You do not have this option 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 `CDOSession` 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.

read() method (Same as fill() method)

Introduced: Mobile release 4.1 See the fill() method for detailed information about the method.

Syntax

```
read ( [parameter-object | filter-string ])
```

parameter-object

An Object initialized with a set of properties that can be used on the server to select, sort, and page the records to be returned. If you are using the JSDO dialect of the Kendo UI DataSource to bind a JSDO instance to Kendo UI widgets, the DataSource initializes and invokes fill( ) with parameter-object. The properties you can initialize when you call fill( ) directly include:

- **filter** — An object containing property settings used to select the records to be returned. These property settings are in the format as the property settings in the Kendo UI DataSouce filter property object. For more information, see the filter configuration property description in the Telerik Kendo UI DataSource documentation.
- **id** — A string specifying a unique ID that the resource understands to identify a specific record.
- **skip** — A number that specifies how many records to skip before returning (up to) a page of data. You must specify this property together with the top property**.**
- **sort** — An expression that specifies how to sort the records to be returned.
- **tableRef** — A string specifying the name of a table reference in the CDO. This property is required when the CDO represents a multi-table resource and the filter property Object is **also** specified with filter information.
- **top** — A number that specifies how many records (the page size) to return in a single page of data after using skip. You must specify this property together with the skip property. The final page of a larger result set can contain a smaller number of records than top specifies.

Note: You must specify both skip and top to implement server paging. If these properties are not specified together, the resource Read operation determines the records to return without using any paging information.

`filter-string`

A string that can be used on the Cloud Data Server to select records to be returned, much like the `WHERE` option of the record phrase. The actual format of this string and its affect on the records returned is determined by the routine on the Cloud Data Server that uses it. For example, you might pass:

- A single key value (e.g., `"30"`)
- A relational expression (e.g., `"CustNum > 10 AND CustNum < 30"`)
- An actual `WHERE` string (e.g., `'Item.CatDescription CONTAINS "ski & (gog* ! pol*)"'`)
- A string using the JSON Filter Pattern

Note: The CDO requires the URI for the `"read"` operation of the resource to contain the following query string: `"?`*`filter`*`=~{`*`filter`*`~}"`, where *`filter`* is the name of a string input parameter defined for the routine that implements the operation (`INPUT` *`filter`* `AS CHARACTER`).

Caution: Using an actual `WHERE` string for a dynamic query can create a potential security issue.

If you do not specify *`filter-string`*, the records returned, again, depend on the routine.

This method invokes the single resource operation that is annotated in the Data Object interface with the `"read"` operation type. The result of calling this method replaces any prior data in CDO memory with the record objects returned by the built-in read operation. These record objects are stored in CDO tables that correspond to the source temp-tables on the Cloud Data Server. If the CDO is accessing a ProDataSet and the `writeDataSetBeforeImage` annotation for the Data Object read operation is set to `true`, the CDO also updates the state of its CDO memory with any before-image data sent with the loaded record objects.

Caution: If the CDO has pending record changes from the client that you want to save on the Cloud Data Server, do not call this method before you call the CDO `saveChanges( )` method. Otherwise, the pending changes will be lost when CDO memory is initialized with records from the Data Object read operation.

Note: After this method initializes CDO memory with record objects, and if you have set up automatic sorting using the `autoSort` property, the record objects of each affected table reference are sorted in CDO memory according to the sort order you have established. If sorting is done using sort fields, any

`String` fields are compared according to the value of the `caseSensitive` property on a given table reference. If the `autoSort` property setting is `false` for a given table reference, its record objects are loaded in the order that they are serialized from the corresponding temp-table on the Cloud Data Server (by its primary key).

This method always executes asynchronously, and fires the following CDO named events, shown in operational order:

beforeReadl event

afterRead event

After this method completes execution, you can read the record objects of CDO memory by using the `find( )`, `findById( )`, `foreach( )`, and `getData( )` methods of the CDO. You can return the schema for this data by using the `getSchema( )` method. You can create a new record object in CDO memory using the CDO `add( )` method, and you can update or delete a single record object in CDO memory by using the `assign( ), update(), or remove( )` method, respectively.

You can merge data returned by an invocation method with the data in CDO memory using the `addRecords( )` method.

Note: OpenEdge initializes every record object with an internal id returned by the getId() method, which uniquely identifies each record in CDO memory.

Note that the value returned by getId() for any given record object can change with each invocation of the `read( )` method.

Note: If a ProDataSet returned from the Cloud Data Server contains before-image data, the state of CDO memory includes the changes to record objects recorded in the before-image data.

Example

The following code fragment shows the `read( )` method invoked on a CDO for a ProDataSet resource (`dsCustomerOrder`):

```
dataSet = new progress.data.CDO( 'dsCustomerOrder' );
dataSet.read();
```

readLocal( ) method

Clears out the data in CDO memory and replaces it with all the data stored in a specified local storage area, including any pending changes and before-image data, if they exist.

Return type: `Boolean`

Syntax

```
readLocal ( [ storage-name ] )
```

`storage-name`

The name of the local storage area whose data is to replace the data in CDO memory. If *`storage-name`* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `cdo_`*`serviceName`*`_`*`resourceName`*, where *`serviceName`* is the name of the Cloud Data Service that supports the CDO instance, and

`resourceName` is the name of the resource (table, dataset, etc.) for which the CDO instance is created.

This method returns `true` if it successfully reads the data from the local storage area; it then replaces CDO memory with this data. If the storage area has no data (is empty), this clears CDO memory instead of replacing it with any data, and the method also returns `true`. If *`storage-name`* does not exist, but otherwise encounters no errors, the method ignores (does not clear) CDO memory and returns `false`. If the method does encounter errors (for example, with the data in the specified storage area), it also leaves CDO memory unchanged and throws an exception.

You can call the CDO `saveChanges( )`, `acceptChanges( )`, or `rejectChanges( )` method after calling this method, and any changes read into CDO memory from local storage are handled appropriately.

Example

The following code fragment replaces the data in CDO memory with all the data currently stored in the default storage area:

```
dataSet = new progress.data.CDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
.
.
.
dataSet.readLocal();
```

record property

A property on a CDO table reference that references a `CloudDataRecord object` with the data for the working record of a table referenced in CDO memory.

If no working record is set for the referenced table, this property is `undefined`.

Data type: CloudDataRecord object

Access: Read-only

The table reference that provides this property can either be the value of a property on the CDO with the name of a referenced table in CDO memory or a reference to the CDO itself if the CDO references only a single table.

The field values and additional OpenEdge properties for the working record are provided by the `data` property of the `CloudDataRecord object` returned by the `record` property.

The `record` property is available only for the following CDO events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

rejectChanges( ) method

Rejects changes to the data in CDO memory for the specified table reference or for all table references of the specified CDO.

If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method applies only when the CDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method **after** calling the `saveChanges( )` method in order to cancel a series of changes that have failed on the Cloud Data Server. If the `autoApplyChanges` property is `true`, the CDO automatically accepts or rejects changes for the specified table reference, or for all table references of the specified CDO, based on the success of the corresponding Data Object record-change operations.

Note: Rejecting all pending changes in CDO memory—or even pending changes for a single table reference—because only some were unsuccessful on the Cloud Data Server might be too broad an action for your application. If so, consider using `rejectRowChanges( )` to reject changes a single table record at a time. For more information, see the description of `rejectRowChanges( )` method.

Return type: `Boolean`

Syntax

```
cdo-ref.rejectChanges ( )
cdo-ref.table-ref.rejectChanges ( )
```

`cdo-ref`

A reference to the CDO. If you call the method on *`cdo-ref`*, the method rejects changes for all table references in the CDO.

`table-ref`

A table reference on the CDO. If you call the method on *`table-ref`*, the method rejects changes for the specified table reference.

When you reject changes on a table reference, this method backs out all pending changes to the record objects for the specified table in CDO memory, and uses the before-image data to return each record to its original data values before the pending changes were made. When you reject changes on the CDO reference, the method backs out all pending changes to the record objects for all table references in CDO memory, and uses the before-image data to return each record to its original data values before the pending changes were made. As the specified changes are rejected, the method also empties any associated before-image data, clears all associated settings of the `getErrorString() method`, and removes the associated pending record change indications from CDO memory.

Note: After this method rejects changes, and if you have set up automatic sorting using the `autoSort` property, all the record objects for affected table references are sorted accordingly. If the sorting is done using sort fields, any `String` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have already successfully applied these changes on the Cloud Data Server using the `saveChanges( )` method, **do not** invoke this method if you want the affected client data to be consistent with the corresponding data on the Cloud Data Server.

Example

The following code fragment shows a CDO created so it **does not** automatically accept or reject changes to data in CDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a handler for the CDO `afterSaveChanges` event to determine if all changes to the `eCustomer` table in CDO memory should be accepted or rejected based on the success of all Data Object create, update, and delete operations on the Cloud Data Server. To change the data for a record, a jQuery event is also defined on an update button to update the corresponding `eCustomer` record in CDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.CDO( { name: 'dsCustomerOrder',
                                                 autoApplyChanges : false }
);
dataSet.eCustomer.subscribe('afterSaveChanges', onAfterSaveCustomers, this);

$('#btnUpdate').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.assign();
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnSave').bind('click', function(event) {
  dataSet.saveChanges();
});

function onAfterSaveCustomers(cdo, success, request) {
  if (success)  {

    // Only want to accept changes if all row changes were successful
   cdo.eCustomer.acceptChanges();

  }
  else  {
    // If any row change was not successful, reject all changes
    cdo.eCustomer.rejectChanges();
    // Additional actions associated with rejecting the pending changes...
  }
}
```

When the update button is clicked, the event handler uses the `findById( )` method to find the original record (`record`) with the matching internal record ID (`#id`) and invokes the `assign( )` method on `record` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete `eCustomer` records and add new `eCustomer` records.

An additional jQuery event also defines a save button that when clicked invokes the `saveChanges()` method to apply all pending changes in CDO memory to the Cloud Data Server. After the method completes, and all results have been returned to the client from the Cloud Data Server, the CDO `afterSaveChanges` event fires, and if any Data Object operations on the Cloud Data Server were **not** successful, the handler calls `rejectChanges( )` to reject all pending `eCustomer` changes in CDO memory. Note that success is returned as true if all the row changes were successful, and false if at least one of the row changes was not successful.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each Data Object record-change operation, one record at a time, across the network. You can also have `saveChanges( )` send all pending record change operations across the network in a single Data Object submit operation. For more information and an example, see the description of the `saveChanges( )` method.

rejectRowChanges( ) method

Rejects changes to the data in CDO memory for a specified record object.

This can be the working record of a table reference or the record specified by a `CloudDataRecord object` reference. If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method applies only when the CDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method for an unsuccessful Data Object record-change operation in the handler for the corresponding CDO event fired in response to executing the `saveChanges( )` method. If the `autoApplyChanges` property is `true`, the CDO automatically accepts or rejects changes to the record object based on the success of the corresponding Data Object operation on the Cloud Data Server.

Return type: `Boolean`

Syntax

```
record-ref.rejectRowChanges ( )
cdo-ref.rejectRowChanges ( )
cdo-ref.table-ref.rejectRowChanges ( )
```

`record-ref`

A reference to a `CloudDataRecord object` for a table reference in CDO memory. You can obtain a `CloudDataRecord object` by:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`, `afterDelete`, or `afterDelete` event.

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the CDO that has a working record.

When you reject changes on a specified record object, this method makes the record reflect all pending changes in CDO memory. As the specified changes are rejected, the method also empties any associated before-image data, clears any settings associated `with the getErrorString() method`, and removes the associated pending change indications from CDO memory.

Note: After this method rejects changes on a record, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `String` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have successfully applied these CDO changes to the Cloud Data Server using the `saveChanges( )` method, **do not** invoke this method if you want the affected client data to be consistent with the corresponding data on the Cloud Data Server.

Example

The following code fragment shows a CDO created so it **does not** automatically accept or reject changes to data in CDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single handler for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in CDO memory should be accepted or rejected based on the success of the corresponding Data Object operation on the Cloud Data Server. To change the data for a record, a jQuery event is also defined on a save button to update the corresponding eCustomer record in CDO memory with the current field values entered in a customer detail form (#custdetail):

```
dataSet = new progress.data.CDO( { name: 'dsCustomerOrder',
                                                  autoApplyChanges : false }
);
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnSave').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.assign();
  dataSet.saveChanges();
});

// Similar controls might be defined to delete and create eCustomer records...

function onAfterCustomerChange(cdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
  }
  else
  {
    record.rejectRowChanges();
    // Perform other actions associated with rejecting this record change
  }
}
```

When the button is clicked, the event handler uses the findById( ) method to find the original record with the matching internal record ID (#id) and invokes the assign( ) method on record with an empty parameter list to update its fields in eCustomer with any new values entered into the form. It then calls the saveChanges( ) method to invoke the Data Object update operation to apply these record changes to the Cloud Data Server. You might define similar events and controls to delete the eCustomer record or add a new eCustomer record.

After each Data Object operation for a changed eCustomer record completes, results of the operation are returned to the client from the Cloud Data Server, and the appropriate event fires. If the operation was **not** successful, the handler calls rejectRowChanges( ) to reject the record change associated with the event in CDO memory. An advantage of using an event to manually reject a record change is that you can perform other actions associated with rejecting this particular change, such as displaying an alert to the user that identifies the error that caused the rejection.

Note: This example shows the default invocation of saveChanges( ), which invokes each Data Object operation, one record at a time, across the network. You can also have saveChanges( ) send all pending record change operations across the network in a single Data Object submit operation. For an example, see the description of the saveChanges( ) method.

remove( ) method

Deletes the specified table record referenced in CDO memory.

The specified record can either be the working record of a referenced table or any record provided by a `CloudDataRecord object`.

After execution, any working record for an associated table, and for any child tables is not set. To synchronize the change on the Cloud Data Server, call the `saveChanges( )` method.

Return type: `Boolean`

Syntax

```
record-ref.remove ( )
cdo-ref.remove ( )
cdo-ref.table-ref.remove ( )
```

`record-ref`

A reference to a `CloudDataRecord object` for a table

record in CDO memory. You can obtain a

`CloudDataRecord object` by:

- Invoking a CDO method that returns record objects from a CDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a CDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a CDO `afterCreate`,
  `afterDelete`, or `afterDelete` event.

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the CDO that has a working record.

Note: This method does not trigger an automatic sort and has no effect on any existing sort order established for the table reference. However, if there is a sort order that depends on the presence or absence of the record object you are removing, and you want to establish a new sort order with this record object absent, you must manually sort the remaining record objects using the `sort( )` method by passing it the same sort function that you used to establish the previous sort order.

Example

The following code fragment shows a jQuery event defined on a delete button to delete the record displayed in a customer detail form from the `eCustomer` table referenced in CDO memory:

```
dataSet = new progress.data.CDO( 'dsCustomerOrder' );

$('#btnDelete').bind('click', function(event) {
  var record = dataSet.eCustomer.findById($('#custdetail #id').val());
  record.remove();
  dataSet.saveChanges();
});
```

The form has been previously displayed with values from the same record. When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`record`) and invokes the `remove( )` method on `record` to delete the record from `eCustomer`.

response property

Returns an object where properties contain data from a Data Object built-in or invoke operation executed on the Cloud Data Server.

Data type: `Object`

Access: Read-only

If a built-in operation (create, read, update, or delete) returns successfully and the response is valid JSON that can be converted to a JavaScript object, the `response` property is a reference to the table or ProDataSet object that is returned from the Cloud Data Server. If the server response is not valid JSON, the `response` property is undefined.

If an invoke operation returns successfully and has no return value or output parameters, the property is `null`. If the invoke operation has a return value, you can read it as the value of the object `_retVal` property. If the operation has output parameters, you can read these parameters as the values of object properties whose case-sensitive names and data types match the names and data types of the output parameters specified for the operation on the Cloud Data Server.

If the operation returns an error, the object contains the following properties:

- **`_retVal`** — A `String` with the value of any `RETURN ERROR` string or

ReturnValue
property for a thrown AppError object

- **_errors** — An array of JavaScript objects, each of which contains two properties: _errorMsg with the error message string and _errorNum with the error number, for one of possibly many -returned errors

  Note: In the current OpenEdge release, this array always returns one object only for the first error (the equivalent of ERROR-STATUS:GET-MESSAGE(1) in).

The response property is available only for the following CDO events:

- afterCreate
- afterDelete
- afterFill
- afterInvoke
- afterUpdate

This request object property is also available for any session online and offline events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (Session object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the online and offline events of the Session object that manages Cloud Data Services for the CDO. The object is also returned as the value of any CDO invocation method that you execute synchronously.

saveChanges( ) method

Synchronizes to the server all record changes (creates, updates, and deletes) pending in CDO memory for the current Data Object resource since the last call to the fill( ) or saveChanges( ) method, or since any prior changes have been otherwise accepted or rejected.

This method always executes asynchronously and returns results (either or both) in subscribed CDO event callbacks or in callbacks that you register through the usage of a concurrency object.

After execution of this method, no working record is set in any tables referenced by the CDO.

Syntax

```
saveChanges ( [ use-submit ] )
```

use-submit

An optional boolean parameter that when false (or not specified), tells saveChanges( ) to individually invoke all pending resource Create, Update, and Delete (CUD) operations, which are sent one record at a time across the network. You can use this option with a CDO that accesses either a single-table or multi-table resource, with or without before-image support. Results for each record change are returned across the network to the CDO after the operation on that record completes.

When true, this parameter tells saveChanges( ) to invoke a single Submit operation on the resource that handles any pending record-change (CUD) operations in a single network request.

Default CUD operation execution: one changed record at a time

Without *use-submit*, the saveChanges( ) method invokes the pending CUD operations in CDO memory one record at a time across the network, and in the following general order of operation type:

1.**"Delete"** — All record deletions are applied.

2.**"Create"** — The creation of all new records is applied.

3.**"Update"** — Updates are applied to all modified records.

The sending of changes for multiple operations on the **same** record is optimized so the fewest possible changes are sent to the server. For example, if a record is first updated, then deleted in CDO memory, only the deletion is sent to the server. However, note that all the changes to the record are applied to CDO memory in a pending manner in case the deletion fails on the server.

Note: Without *use-submit*, this method performs no batching of resource CUD operations. That is, the Delete operation is invoked over the network for each deleted record, followed by the Create operation for each created record, and finally by the Update operation for each updated record. So, even for a ProDataSet resource that supports before-imaging, each CUD operation executes over the network only one record at a time and cannot be part of a multi-record transaction.

Submit operation execution: multiple changed records at a time

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the Mobile or Web application:

1.**Using CDO named events for all environments** — The following events fire before and after the saveChanges( ) method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:

a.**beforeSaveChanges event**

b.**beforeDelete event**

c.**afterDelete event**

d.**beforeCreate event**

e.**afterCreate event**

f.**beforeUpdate event**

g.**afterUpdate event**

h.**afterSaveChanges event**

2.**Using a Concurrency Object such as a jQuery Promise object in JavaScript** — Any callbacks that you register using Concurrency Object methods all execute
both **after** the saveChanges( ) method itself and **after** all subscribed CDO after* event callbacks complete execution. Note that the signatures of the callback method for a Concurrency Object matches the signature of the afterSaveChanges event callback function so you can specify an existing afterSaveChanges event callback as the callback function that you register using any Concurrency Object method.

Because the callbacks that you register with any returned Concurrency Object methods execute only after all subscribed after* event callbacks complete, you can invoke logic in registered Concurrency Object method callbacks that can modify any processing done by the event callbacks.

Note that your programming requirements for any afterCreate, afterUpdate, afterDelete (CUD operation) event callback, and for any afterSaveChanges event or Concurrency Object method callback can be affected by your setting of the JSDO autoApplyChanges property.

Behavior of event and Concurrency Object callbacks when autoApplyChanges is true

If you set the autoApplyChanges property to true (the default setting) before you
invoke saveChanges(false), and a corresponding record-change (CUD) operation succeeds on the server, the JSDO automatically accepts and synchronizes the change in CDO memory.

If any CUD operation fails, the operation is automatically undone and the CDO rejects the change by reverting the applicable record in CDO memory to its last-saved state. Specifically:

- If a Create operation fails, the record is removed from CDO memory.
- If an Update operation fails, the record reverts to the state it was in immediately preceding the assign( ) method invocation that led to the failure.
- If a Delete operation fails, the record is restored to CDO memory in its last-saved state. This state does not reflect any unsaved assign( )method invocations that may have occurred before the remove( ) call.

When the CDO synchronizes CDO memory for CUD operations, it uses any before-image data (if available) in each response from invoking the saveChanges( ) method.

If you invoke a Submit operation (saveChanges(true)) on an OpenEdge temp-table resource and leave autoApplyChanges set to true, the CDO throws and exception, because it cannot automatically synchronize CDO memory with the results of a Submit operation on a resource that has no before-imaging support. To invoke Submit on a resource without before-imaging, you must first set autoApplyChanges to false, and handle the synchronization of CDO memory according to a custom data management model. For more information, see the section on behavior when autoApplyChanges is false in this saveChanges( ) method description.

If you invoke a Submit operation (saveChanges(true)) on an OpenEdge ProDataSet resource with before-image support, CDO memory is automatically updated or not based on the success or failure of each record create, update, and delete included in the Submit operation request.

Note: For a Submit operation on a before-image resource, if you want all such record changes rejected if even one record change returns an error, set autoApplyChanges to false and explicitly reject all record changes based on the returned error condition. For more information, see the section on behavior when autoApplyChanges is false in this saveChanges( ) method description.

When autoApplyChanges is true, the CDO automatically clears any error conditions set for the affected record changes only **after** the record changes have all been rejected and undone and **after** all registered after* event and Concurrency Object method callbacks have executed. For more information, see the section on error handling in this saveChanges( ) method description.

Note: You can manually inspect error conditions and information as part of a callback execution when autoApplyChanges is true. However, checking for such error information might not be as useful as when autoApplyChanges is false, because all CDO memory changes associated with record-change errors are automatically undone.

Caution: Use care when taking any action within event or Promise method callbacks that might interfere with the automatic acceptance or rejection of pending record changes. If you want to manually manage the handling of pending record changes in response to resource operations in these callbacks, consider setting autoApplyChanges to false before invoking saveChanges( ).

Behavior of event and Promise callbacks when autoApplyChanges is false

If you set the autoApplyChanges property to false before you invoke saveChanges( ), you must use one of the following methods to manually accept or reject any record changes in order to synchronize CDO memory with operation results from the server. Use the method that is appropriate for the resource operation and the CDO event or Concurrency Object method callback where you manage these changes:

- acceptChanges( )

- acceptRowChanges( )
- rejectChanges( )
- rejectRowChanges( )

Depending on the success of the particular resource operation and the CDO after* event or Concurrency Object method callback in which you respond to operation results, you can check returned request object properties to determine what accept*( ) or reject*( ) method to call. If you are handling results for one or more record-change (CUD) operations in an afterSaveChanges event callback, or in a Concurrency Object method callback, in response to calling saveChanges(false) (without using Submit), you can use the batch property of the request object to evaluate the results of each CUD operation invoked by saveChanges( ).

If you are handling results of a Submit operation (saveChanges(true)) on an OpenEdge temp-table resource **without** before-imaging, and the operation otherwise completes successfully, in an afterSaveChanges event or Promise method callback you can inspect both the response and jsrecords properties of the returned request object to evaluate the results and either accept or reject each record change in CDO memory, or manually update CDO memory further before accepting the additional record changes in CDO memory.

Note: After a Submit operation completes on a resource without before-image support, the response property contains the updated record objects returned from the server and the jsrecords property contains the record objects sent to the server with changes for the request, and none of these record objects contain record ID information. You must therefore compare the record objects returned in the response property with those sent in the jsrecords property to determine how to manage the Submit operation results according to any custom data management model established for the resource on the server.

If you want to evaluate the results of any single record-change operation invoked by saveChanges(false) (a CUD operation with or without before-image support), or saveChanges(true) (a Submit operation) **with** before-image support, you can inspect the request object returned in an appropriate after* event callback that you have registered for each CUD operation, where you can identify the associated record change using the jsrecord property.

Note that the "acceptChanges()" and rejectChanges()" methods might not be as useful for synchronizing CDO memory with the server as the corresponding "acceptRowChanges()" and "rejectRowChanges()" methods, which you can invoke selectively in response to the results of individual record-change operations. The "acceptChanges( )" and "rejectChanges()" methods are most useful for accepting or rejecting all record changes in CDO memory based on the results of a single server transaction (invoked using Submit with before-image support) that either succeeds and applies all server record changes or fails if even one record change fails and undoes all server record changes as part of undoing the transaction. For more information, see the reference description for each method.

For more information on handling errors from failed record-change operations, see the information on error handling later in this "saveChanges()" method description.

Example

The following code fragment shows a JSDO created for an OpenEdge ProDataSet resource with before-imaging, and it **does not** automatically accept or reject changes to data in JSDO memory after a call to the saveChanges(true) method. Instead, it subscribes a single callback for each of the afterDelete, afterCreate, and afterUpdate, events to determine if changes to any eCustomer table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is defined on an update button to update the corresponding eCustomer record in JSDO memory with the current field values entered in a customer detail form (#custdetail):

Example: saveChanges(*true*) // using Submit

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnUpdate').bind('click', function(event) {
    var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
    jsrecord.assign();
});

// Similar controls might be defined to delete and create eCustomer records...
$('#btnCommit').bind('click', function(event) {
   dataSet.saveChanges(true); // Invokes the resource Submit operation
});

function onAfterCustomerChange(jsdo, record, success, request) {
 if (success)  {
   record.acceptRowChanges();
   // Perform other actions associated with accepting this record change
 }
 else
 {

     /* check for OpenEdge errors on a record change */
    if (request.response && request.response._errors &&
        request.response._errors.length > 0) {
        var lenErrors = request.response._errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            var errorEntry = request.response._errors[idxError];
             var errorMsg = errorEntry._errorMsg;
             var errorNum = errorEntry._errorNum;
             /* handle update OpenEdge error . . . */
         }
   }
   /* call getErrorString() on the changed record to return error string associated with before-image
    * data and handle the error */
   var jsrecError = record.getErrorString();
    if (jsrecError) {
         record.rejectRowChanges();
         // Perform any other actions associated with rejecting record change
   }



  }
}
```

When the button is clicked, the event handler uses the findById( ) method to find the original record with the matching internal record ID (#id) and invokes the assign( ) method on jsrecord with an empty parameter list to update its fields in eCustomer with any new values entered into the form. You might define similar events and controls to delete the eCustomer record or add a new eCustomer record.

A jQuery event also defines a commit button that when clicked invokes the saveChanges( ) method, passing true as the *use-submit* parameter value, to apply all pending changes in JSDO memory on the server in a single network request. Using this parameter, all pending record deletes, creates, and updates, including before-image data, are sent to the server in a single ProDataSet as input to a Submit operation. This operation processes all the changes for each record delete, create, or update on the server, storing the results in the same ProDataSet, including any errors, for output to the client in a single network response.

After the method completes, and all results have been returned to the client from the server, the appropriate event for each resource Delete, Create, or Update operation fires even though multiple changes can be sent using a single Submit operation. If the operation **was** successful, the event handler calls acceptRowChanges( ) to accept the eCustomer record change associated with the event in JSDO memory. If the operation **was not** successful, the event handler calls rejectRowChanges( ) to reject the eCustomer record change. An advantage of using an event to manually accept or reject a record change is that you can perform other actions associated with this particular change, such as creating a local log that describes the change or reports the error.

Example: saveChanges( ) on an OpenEdge resource using a Promise object

Note: submit parameter is not specified, so operations are sent to backend one at a time across network.

```
/* some code that initiates multiple CRUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
jsrecord.remove();

/* call to saveChanges() with inline-coded Promise callback handling */
myjsdo.saveChanges().done(
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<field-ref> */
            . . .
        });
    }).fail(
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        /* number of operations invoked */
        var len = request.batch.operations.length;

        for(var idx = 0; idx < len; idx++) {
            var operationEntry = request.batch.operations[idx];

            console.log("Operation: " + operationEntry.fnName);
            console.log("Operation code: " + operationEntry.operation)

            if (!operationEntry.success) {
              /* handle operation error condition */
              if (operationEntry.response && operationEntry.response._errors &&
                    operationEntry.response._errors.length > 0) {
```

```
            var lenErrors = operationEntry.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

              var errors = operationEntry.response._errors[idxError];
              var errorMsg = errors._errorMsg;
              var errorNum = errors._errorNum;
              /* handle error results */
              . . .
            }
          }
        }
        else {
          /* operation succeeded */
        }
      }
    });
```

In the above example, the same processing occurs for successful and unsuccessful execution of saveChanges( ), but there is no need to test the success parameter of the done( ) and fail( ) Promise method callbacks, because they each execute in response to this value.

However, if your code already uses the afterSaveChanges event implementation, and you want to quickly change it to use Promises, you can simply register the original onAfterSaveChanges function as the callback for the always( ) Promise method

saveLocal( ) method

Saves CDO memory to a specified local storage area, including pending changes and any before-image data, according to a specified data mode.

Return type: `undefined`

Syntax

```
saveLocal ( [ storage-name [ , data-mode ] ] )
saveLocal ( data-mode )
```

storage-name

The name of the local storage area in which to save the specified data from CDO memory. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `cdo_`*`serviceName`*`_`*`resourceName`*, where *serviceName* is the name of the Cloud Data Service that supports the CDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the CDO instance is created.

data-mode

A CDO class constant that specifies the data in CDO memory to be saved to local storage. Each data mode initially clears the specified local storage area of all its data, and then replaces it with the data from CDO memory as specified. The possible values include:

- **`ALL_DATA`** — Replaces the data in the storage area with **all** the data from CDO memory, including pending changes and any before-image data. This is the default data mode.
- **`CHANGES_ONLY`** — Replaces the data in the storage area with **only** the pending changes from CDO memory, including any before-image data.

If this method encounters any errors, it leaves the specified storage area unchanged and throws an exception.

This method supports any schema that the CDO supports.

If you want to save CDO memory to local storage after the CDO `saveChanges( )` method fails in response to an offline condition, be sure to set the `autoApplyChanges` property on the CDO to `false` before calling this method for the first time. You can then continue to save CDO memory to protect against a local session failure as it accumulates further offline changes until the Mobile or Web application goes back online and `saveChanges( )` succeeds in saving the changes to the server.

You can also use this method to routinely cache static data, such as state and rate tables, that might not change often, allowing for faster startup of the Mobile or Web application. One way to do this is to define a CDO for a resource that accesses only static data, and invoke this method after refreshing CDO memory using the `fill( )` method. When caching data in general, be sure to save CDO memory when it is in a consistent state, such as immediately after successful invocation of the CDO `fill( )` or `saveChanges( )` method (in the relatively rare case where routinely static data is being updated).

Example

The following code fragment caches CDO memory for a CDO defined with static data immediately after it is loaded into CDO memory:

```
dataSet = new progress.data.CDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
```

In this case, all the data in CDO memory is cached to the default storage area.

services property

Returns an array of objects that identifies the Cloud Data Services that have been loaded for the current `CDOSession` object and its Mobile or Web application.

Data type: `Object` array

Access: Read-only

You load Cloud Data Services for a CDO`Session` object by loading the corresponding CDO catalogs using the `addCatalog( )` method after you login using the `login( )` method.

Each object in the array contains two properties:

- **`name`** — The name of a Cloud Data 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 CDO`Session` object's `serviceURI` property, which contains the Mobile or Web application URI passed to the `login( )` method.

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

Example

Given the following service names and URIs loaded for a CDO`Session` object:

- **`"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 CDOSession(
                   { serviceURI: 'http://custhost:8080/SportsApp');

 // log in pdsession.login();

 // load 2 catalogs
 pdsession.addCatalog
   ("http://custhost:8080/SportsApp/static/mobile/CustomerSvc.json");
  pdsession.addCatalog
   ("http://custhost:8080/SportsApp/static/mobile/ItemSvc.json");

 // Use services property to print services loaded by this Session object
 for (var i=0; i < pdsession.services.length; i++) {
   console.log( pdsession.services[i].name + "   "
                + pdsession.services[i].uri );
 }
```

Output from the preceding code fragment:

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

serviceURI property

Returns the Mobile or Web application URI that was passed to the class constructor for the current CDOSession object, whether or not the most recent call to `login( )` succeeded.

Data type: `String`

Access: Read-only

setSortFields( ) method

Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).

This method enables or disables automatic sorting based on record fields only for supported CDO operations. See the description of the `autoSort` property for more information.

After completing execution, this method has no effect on the working record for the affected table reference.

Return type: `null`

Syntax

```
cdo-ref.setSortFields ( sort-fields )
cdo-ref.table-ref.setSortFields ( sort-fields )
```

cdo-ref

A reference to the CDO. You can call the method on *cdo-ref* if the CDO has only a single table reference.

table-ref

A table reference on the CDO.

sort-fields

An array of `String` values set to the names of record fields on which to sort the record objects, with an optional indication of the sort order for each field. This array can have the following syntax:

Syntax:

```
[ "field-name[:sort-order]" [ , "field-name[:sort-order]" ] ...
 ]
```

field-name

The name of a field in the record objects of the specified table reference. A *field-name* can have the OpenEdge-reserved name, `"_id"`. Otherwise, any *field-name* must already exist in the CDO schema and must have a scalar value (cannot be an array field).

sort-order

An indication of the sort order for the field, which can have one of the following case-insensitive values:

- `ASC` — Sorts ascending.
- `ASCENDING` — Sorts ascending.
- `DESC` — Sorts descending.
- `DESCENDING` — Sorts

descending. The default

sort order is ascending.

When the automatic sort occurs, the record objects are sorted and grouped by each successive *`field-name`* in the array, according to its JavaScript data type and specified *`sort-order`*. Fields are compared using the >, <, and = JavaScript operators. `String` fields can be compared with or without case sensitivity depending on the `caseSensitive` property setting. However, note that date fields are compared as dates, even though they are represented as strings in JavaScript.

If you set the *`sort-fields`* parameter to `null`, or you specify an empty array, the method clears all sort fields. Automatic sorting for the table reference can then occur only if there is an existing sort function setting using the `setSortFn( )` method.

Note: If you set a sort function for the table reference using `setSortFn( )` in addition to using this method to set sort fields, the sort function takes precedence.

Example

In the following code fragment, assuming the `autoSort` property is set to `true` on `dsCustomer.eCustomer` (the default), after the `fill( )` method initializes CDO memory, the record objects for `eCustomer` are sorted by the `Country` field ascending, then by the `State` field within `Country` ascending, then by the `Balance` field within `State` descending. At a later point, the `foreach( )` method then loops through these record objects, starting with the first record in `eCustomer` sort order:

```
dsCustomer = new progress.data.CDO( { name: 'dsCustomer' });
dsCustomer.eCustomer.setSortFields( [ "Country", "State", "Balance:DESC" ]
);
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

setSortFn( ) method

Specifies or clears a user-defined sort function with which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).

This method enables or disables automatic sorting based on a sort function only for supported CDO operations. See the description of the `autoSort` property for more information.

After completing execution, this method has no effect on the working record for the affected table reference.

Return type: `null`

Syntax

```
cdo-ref.setSortFn ( funcRef )
cdo-ref.table-ref.setSortFn ( funcRef )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`funcRef`

A reference to a JavaScript sort function that compares two record objects for the sort and returns a `Number` value. This function must have following signature:

Syntax:

```
function [ func-name ] ( record-ref1 , record-ref2 )
```

Where *`func-name`* is the name of a function that you define external to the `setSortFn( )` method parameter list, and *`record-ref1`* and *`record-ref2`* are two `CloudDataRecord object`s that the function compares from the specified table reference. You can then pass *`func-name`* to the `setSortFn( )` method as the *`funcRef`* parameter. Alternatively, you can specify *`funcRef`* as the entire inline function definition without *`func-name`*.

Your *`funcRef`* code determines the criteria by which one of the two input record objects follows the other in the sort order, and returns one of the following values depending on the result:

- **`1`** — The *`record-ref1`* object follows (is "greater than") the *`record-ref2`* object in the sort order.
- **`-1`** — The *`record-ref1`* object precedes (is "less than") the *`record-ref2`* object in the sort order.
- **`0`** — The two record objects occupy the same position (are "equal") in the sort

order.

When OpenEdge calls an automatic sort, and a sort function is set using this method, the sort uses this function to determine the sort order for every pair of records that it tests as it iterates through the record objects of the specified table reference.

If you set the *funcRef* parameter to null, the method clears any sort function definition. Automatic sorting for the table reference can then occur only if there are one or more existing sort fields set using the setSortFields( ) method.

Note: Any default JavaScript comparisons that you make with String fields in *funcRef* are case sensitive according to JavaScript rules and ignore the setting of the caseSensitive property.

Note: If you set sort fields for the table reference using setSortFields( ) in addition to using this method to set a sort function, the sort function takes precedence.

Examples

In the following code fragment, assuming the autoSort property is set to true on dsCustomer.eCustomer (the default), after the fill( ) method initializes CDO memory, the record objects for eCustomer are automatically sorted using the results of the external user-defined function, sortOnNameCSensitive( ), whose reference is passed to the setSortFn( ) method. In this case, the function compares the case-sensitive values of the Name fields from each pair of eCustomer record objects selected by the sort. At a later point, the foreach( ) method then loops through these record objects, starting with the first record in eCustomer sort order:

```
dsCustomer = new progress.data.CDO( { name: 'dsCustomer' });
dsCustomer.setSortFn ( sortOnNameCSensitive );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );

function sortOnNameCSensitive ( rec1 , rec2 ) {
  if (rec1.data.Name > rec2.data.Name)
    return 1;
  else if (rec1.data.Name < rec2.data.Name)
    return -1;
  else    return 0;
  }
```

If you want to compare the Name field in this function using a case-insensitive test, you can use the JavaScript toUpperCase( ) function in the user-defined function. For example, in sortOnNameCInsensitive( ), as follows:

```
dsCustomer = new progress.data.CDO( { name: 'dsCustomer' });
dsCustomer.setSortFn ( sortOnNameCInsensitive );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );

function sortOnNameCInsensitive ( rec1 , rec2 ) {
  if (rec1.data.Name.toUpperCase() > rec2.data.Name.toUpperCase())
    return 1;
  else if (rec1.data.Name.toUpperCase() < rec2.data.Name.toUpperCase())
    return -1;
  else
    return 0;
  }
```

sort( ) method

Sorts the existing record objects for a table reference in CDO memory using either specified sort fields or a specified user-defined sort function.

After completing execution, this method has no effect on the working record for the affected table reference.

Return type: `null`

Syntax

```
cdo-ref.sort ( { sort-fields | funcRef } )
cdo-ref.table-ref.sort ( { sort-fields | funcRef } )
```

`cdo-ref`

A reference to the CDO. You can call the method on *`cdo-ref`* if the CDO has only a single table reference.

`table-ref`

A table reference on the CDO.

`sort-fields`

An array of `String` values set to the names of record fields on which to sort the record objects, with an optional indication of the sort order for each field. This array can have the following syntax:

Syntax:

```
[ "field-name [ :sort-order ] " [ , "field-name [ :sort-order ] " ] . . .
 ]
```

field-name

The name of a field in the record objects of the specified table reference. A *field-name* can have the OpenEdge-reserved name, "_id". Otherwise, any *field-name* must already exist in the CDO schema and must have a scalar value (cannot be an array field).

sort-order

An indication of the sort order for the field, which can have one of the following case-insensitive values:

- ASC — Sorts ascending.
- ASCENDING — Sorts ascending.
- DESC — Sorts descending.
- DESCENDING — Sorts

descending. The default

sort order is ascending.

When the sort occurs, the record objects are sorted and grouped by each successive *field-name* in the array, according to its JavaScript data type and specified *sort-order*. Fields are compared using the >, <, and = JavaScript operators. String fields can be compared with or without case sensitivity depending on the caseSensitive property setting. However, note that date fields are compared as dates, even though they are represented as strings in JavaScript.

funcRef

A reference to a JavaScript sort function that compares two record objects for the sort and returns a Number value. This function must have following signature:

Syntax:

```
function [ func-name ] ( record-ref1 , record-ref2 )
```

Where *`func-name`* is the name of a function that you define external to the `sort( )` method parameter list, and *`record-ref1`* and *`record-ref2`* are two `CloudDataRecord objects` that the function compares from the specified table reference. You can then pass *`func-name`* to the `sort( )` method as the *`funcRef`* parameter. Alternatively, you can specify *`funcRef`* as the entire inline function definition without *`func-name`*.

Your *`funcRef`* code determines the criteria by which one of the two input record objects follows the other in the sort order, and returns one of the following values depending on the result:

- **`1`** — The *`record-ref1`* object follows (is "greater than") the *`record-ref2`* object in the sort order.
- **`-1`** — The *`record-ref1`* object precedes (is "less than") the *`record-ref2`* object in the sort order.
- `0` — The two record objects occupy the same position (are "equal") in the sort order.

When you invoke the `sort( )` method with a sort function, the sort uses this function to determine the sort order for every pair of records that it tests as it iterates through the record objects of the specified table reference.

Note: Any default JavaScript comparisons that you make with `String` fields in *`funcRef`* are case sensitive according to JavaScript rules and ignore the setting of the `caseSensitive` property.

Caution: Because the `sort( )` method executes in JavaScript on the client side, sorting a large set of record objects can take a significant amount of time and make the UI appear to be locked. You might set a wait or progress indicator just prior to invoking the sort to alert the user that the app is working.

Examples

In the following code fragment, the `fill( )` method initializes CDO memory with `eCustomer` record objects from the Cloud Data Server in order of the table primary key (the default). The `sort( )` method later sorts the record objects for `eCustomer` by the `Country` field ascending, then by the `State` field within `Country` ascending, then by the `Balance` field within `State` descending. The `foreach( )` function then loops through these record objects in the new `eCustomer` sort order:

```
dsCustomer = new progress.data.CDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( [ "Country", "State", "Balance:DESC" ] );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

In the following code fragment, the `fill( )` method initializes CDO memory with `eCustomer` record objects from the Cloud Data Server in order of the table primary key (the default). The `sort( )` method later sorts the record objects for `eCustomer` using the results of an inline function definition, which in this case compares the case-sensitive values of the `Name` fields from each pair of `eCustomer` record objects selected by the sort. The `foreach( )` method then loops through these record objects in the new `eCustomer` sort order:

```
dsCustomer = new progress.data.CDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( function( rec1 , rec2 ) {
  if (rec1.data.Name > rec2.data.Name)
    return 1;
  else if (rec1.data.Name < rec2.data.Name)
    return -1;
  else
    return 0;
  } );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

If you want to compare the `Name` fields using a case-insensitive test, you can use the JavaScript

`toUpperCase( )` function in the inline function definition, as follows:

```
dsCustomer = new progress.data.CDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( function
( rec1 , rec2 ) {
  if (rec1.data.Name.toUpperCase() > rec2.data.Name.toUpperCase())
    return 1;
  else if (rec1.data.Name.toUpperCase() < rec2.data.Name.toUpperCase())
    return -1;
  else
    return 0;
  } );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

subscribe( ) method (CDO class)

Subscribes a given event handler function to a named event of the current CDO or table reference.

For more information on these events, see the reference entry for CDO class.

After execution, the working record for any associated table reference remains unchanged.

Return type: `null`

Syntax

```
cdo-ref.subscribe ( event-name [ , op-name ] ,
                    event-handler [ , scope ] )
cdo-ref.table-ref.subscribe ( event-name [ , op-name ] ,
                              event-handler [ , scope ] )
```

cdo-ref

A reference to the CDO. If you call the method on *cdo-ref*, you can subscribe the event handler to any event that fires on the CDO and all its table references.

table-ref

A table reference on the CDO. If you call the method on *table-ref*, you can subscribe the event handler to an event that fires only on the table reference.

event-name

The name of an event to which you subscribe an event handler. See the reference entry for CDO class for a list of available CDO events.

If you call the `subscribe( )` method on *table-ref*, you can subscribe the event handler

only to the following events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`

- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

`op-name`

The name of a CDO invocation method, a call to which causes the event to fire. This parameter is required in cases where *`event-name`* is `beforeInvoke` or `afterInvoke`. Use it **only** with these event names, and only when subscribing on the CDO (not on a *`table-ref`*). The value of `op-name` is the same as the `fnName` property on the request object.

`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).

subscribe( ) method (CDOSession class)

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

Subscribes *event-handler* to *event.* The *event* must be an event defined by the CDOSession API.

The method's return type is undefined. It throws an error if the event is not supported.

<u>Parameters:</u>

event

A string that identifies the event. Case doesn't matter.

eventHandler

Reference to an event handler function.

scope *(optional)*

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).

success property

A `Boolean` that when set to `true` indicates that the Data Object operation was successfully executed.

This property is set from the HTTP status code returned from the server. A successful Data Object operation returns an HTTP status code in the range of 200 - 299. An unsuccessful operation causes a value outside this range to be returned for the HTTP status code and sets this property to `false`.

Data type: `Boolean`

Access: Read-only

The `success` property is available only for the following CDO events:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterSaveChanges`
- `afterUpdate`

In the case of an `afterSaveChanges` event, the `success` property is `true` only if **all**

record-change operations that are invoked by the `saveChanges( )` method were successfully executed.

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (`Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO. The object is also returned as the value of any CDO invocation method that you execute synchronously.

Note: When the Cloud Data Server routine that implements a Data Object operation raises an unhandled error, this causes an HTTP status code of 500 and any included error information to be returned from the server. This can occur when either a routine raises an application error or the virtual machine (AVM) raises a system error, and the error is thrown out of the top-level Cloud Data Server routine.

table reference property (CDO class)

An object reference property on a CDO that has the name of a table mapped by the resource to a table for which the current CDO is created.

Its value is a reference (*table reference*) to the table object in CDO memory. This table object provides access to a working record, if defined. If a resource maps to a ProDataSet, its CDO provides one table reference for every table in the ProDataSet.

Data type: Table object reference in CDO memory

Access: Read-only

In syntax, wherever a table reference can be used, `table-ref` represents the name of the property containing the table reference. A referenced table object provides the following properties:

- **`record` property**— A reference to a `CloudDataRecord object`, which provides the data for the working record of the table in its `data` property. This `data` property provides access to the field values of the working record as corresponding field reference properties (see the following bullet). If no working record is defined for the table, the `record` property is `null`.
- **field reference property** — Also referred to as a *field reference*, a property on the table object that has the name of a field (as defined in the mapped table schema) and the value for that field in the working record. In syntax, wherever a field reference appears, `field-ref` represents the name of the property containing the field reference. A table object provides one field reference for each field defined in the mapped table. If no working record is defined, all field references are `null`, **except** when fields are referenced on the `data` property of a `CloudDataRecord object` reference.

Caution: Never write directly to a `field-ref` using this `record` property or any `CloudDataRecord object` reference; in this case, use `field-ref` **only to read** the data. Writing field values using a `CloudDataRecord object` reference does **not** mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in CDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a `cdo-ref.table-ref.field-ref` for a working record or by calling the `assign( )` method on a valid `table-ref` or `CloudDataRecord object` reference.

You can therefore reference the field values in the working record of a given table reference using corresponding field references **either** on the `data` property of the `CloudDataRecord object` returned by the `record` property of the table reference or directly on the table reference itself. The `record` property provides an alternative way to access table fields, especially for a table field that has the same name as a CDO property (or method) that you can access (or invoke) directly on a table reference.

Example

For example, the following code fragment shows two different ways to access the `CustNum` field of a record added for a `customer` table object mapped by a ProDataSet resource:

```
var dataSet = new Progress.data.CDO( 'CustomerOrderDS' );
dataSet.customer.add();
alert(dataSet.customer.record.data.CustNum);
alert(dataSet.customer.CustNum);
```

Both calls to the `alert( )` function access the same `CustNum` field in the working record of the `customer` table created by the `add( )` method.

For more information on accessing the working record of a table reference, see the notes in the section on CDO class.

unsubscribe( ) method (CDO class)

Unsubscribes a given event handler function from a named event of the current CDO or table reference.

For more information on these events, see the refernce entry for CDO class.

After execution, the working record for any associated table reference remains unchanged.

Return type: `null`

Syntax

```
cdo-ref.unsubscribe ( event-name [ , op-name ] ,
                      event-handler [ , scope ] )
cdo-ref.table-ref.unsubscribe ( event-name [ , op-name ] ,
                                event-handler [ , scope ] )
```

cdo-ref

A reference to the CDO. If you call the method on *cdo-ref*, you can unsubscribe the event handler from any event that fires on the CDO and all its table references.

table-ref

A table reference on the CDO. If you call the method on *`table-ref`*, you can unsubscribe the event handler from an event that fires only on the table reference.

`event-name`

The name of an event to which you unsubscribe an event handler. See the refernce entry for CDO class for a list of available CDO events.

If you call the `unsubscribe( )` method on *`table-ref`*, you can unsubscribe the event handler **only** from the following events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

`op-name`

The name of a CDO invocation method, a call to which causes the event to fire. This parameter is required in cases where *`event-name`* is `beforeInvoke` or `afterInvoke`. Use it **only** with these event names, and only when unsubscribing on the CDO (not on a *`table-ref`*). To be meaningful, the *`op-name`* value must match the corresponding

`op-name` parameter in a preceding `subscribe( )` method call.

`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. 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.

unsubscribe( ) method (CDOSession class)

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

Removes the subscription of *event-handler* to *event.* The method's return type is undefined. It throws an error if the event is not supported.

Parameters:

event

A string that identifies the event. Case doesn't matter.

eventHandler

Reference to the event handler function that is to be unsubscribed.

scope *(optional)*

Specifies the execution scope of the event handler function that is being unsubscribed. If the scope property is omitted, the execution scope is the global object (usually the browser or device window).

unsubscribeAll( ) method (CDO or CDOSession class)

Unsubscribes all event handler functions from a named event of the current CDO or `CDOSession` object, or unsubscribes all handlers from all events of the current CDO or `Session` object.

Return type: `null`

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 CDO class or the CDOSession class for a list of available events.

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

update() method (Same as assign() method)

Introduced: Mobile Release 4.1

Syntax

```
jsrecord-ref.update ( update-object )
jsdo-ref.update ( update-object )
jsdo-ref.table-ref.update ( update-object )
```

Example

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
    var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
    jsrecord.update(update-object);
   dataSet.saveChanges();
});
```

useRelationships property

A `Boolean` that specifies whether CDO methods that operate on table references in CDO memory work with the table relationships defined in the schema (that is, work only on the records of a child table that are related to the parent).

Data type: `Boolean`

Access: Readable/Writable

When set to `true`, methods, such as `add( )`, `find( )`, and `foreach( )`, that have default behavior for related table references respect these relationships when operating on related tables. When set to `false`, these methods operate on all table references as if they have no relationships. The default value is `true`.

userName property

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

Data type: `String`

Access: Read-only

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

Note: This property does not always 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.

xhr property

A reference to the XMLHttpRequest object used to perform a Data Object operation request.

In the case of an asynchronous call, this property may not be available until after the XMLHttpRequest object is created.

Data type: `Object`

Access: Read-only

The `xhr` property is available only for the following events after calling the CDO `saveChanges( )`

method either with an empty parameter list or with the single parameter value of `false`:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterUpdate`

The `xhr` property is available only for the following event after calling `saveChanges(true)` on a CDO enabled for before-image support:

- `afterSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated Data Object operation when it encounters a change in the online status of the CDO's login session (`Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to CDO events and to the `online` and `offline` events of the `Session` object that manages Cloud Data Services for the CDO. The object is also returned as the value of any CDO invocation method that you execute synchronously.

Which methods access the server

The following chart identifies the CDO operations and whether the operation communicates with the server.

CDO Operation	Local	Remote	Description
acceptChanges	✓		Accept all changed records and mark each record as not-edited. Remove the before-image for each record.
acceptRow Changes	✓		Accept the changes to the selected record and mark the record as not edited. Remove the before-image for the record.
add alias: create	✓		To create a new record in local storage on the client, you call the add() method on the table reference on the CDO. The fields of the new record are initialized with the values specified in an object passed to the method. For any fields whose values are not provided in this object default values are taken from schema in the CDO catalog. When the operation is complete, the new record becomes the working record for the associated temp-table. If the temp-table has child temp-tables, the working record for those child tables are not set
addRecords	✓		Reads a json object and updates the local storage of the CDO. The data is merged into CDO local storage and affects existing data according to a specified merge mode and optional key fields
assign alias: update	✓		To modify an existing record in local storage on the client you can call the assign() or update() method. The values of fields to be modified are specified in an object passed to the method. When the operation is complete, any working records previously set before the method executed remain as the working record.
fill alias: read		✓	To load data into local storage on the client, you call the fill() or read() method on the CDO. Each time the method is called all records currently in local storage are cleared and replaced by the records returned by the method. When the operation is complete, the working record for each referenced temp-table is set to its first record, depending on any active parent-child relationships. So, for each child temp-table, the first record is determined by its relationship to the related working record in its parent temp-table.
remove	✓		To delete an existing record from local storage on the client, you call the remove() method. When the operation is complete, any working record for an associated temp-table and any child temp-table are not set.
rejectChanges	✓		Revert all changed records back to their original values before any changes were made since the last fill(), read(), acceptRowChanges() or acceptChanges() method.
rejectRowChanges	✓		Revert a record back to the original values before any changes were made since the last fill(), read(), acceptRowChanges() or acceptChanges() method.

CDO Operation	Local	Remote	Description
saveChanges		✓	Synchronizes to the data source all changes made to CDO memory since the last call to fill(), read() or saveChanges() method. The saveChanges() method completes this data synchronization. The data source handles the ordering of the changes by invoking appropriate build-in resource operations in the following general order of operation type: 1. Delete – All record deletions are applied 2. Create – The creation of all new records is applied 3. Update – Updates are applied to all modified records The sending of changes for multiple operations on the same record is optimized so the fewest possible changes are sent to the AppServer. For example, if a record is updated then deleted in local storage, only a delete request is sent to the data source. After execution the working record for each temp-table referenced by the CDO is not set. autoApplyChanges is a flag set at the CDO level. When set to true (the default), acceptChanges() or rejectChanges() are automatically called beased on whether the change succeeded or returned an error. You can still access the error returned. To override the behavior, set this flag to false.