ABL JDBC Business Logic Driver

Table of contents

Introduction

ABL JDBC

The standard interface that glues Progress Business Logic and Java.

ABL JDBC driver brings the power of Progress ABL to the Java world in a standardized way. By leveraging your existing business logic the ABL JDBC driver can expose it to powerful reporting engines or data integration tools. This can help you to integrate versatile reporting solutions inside your Progress ABL application or easily respond to application and data integration requests without the pain of going to complex application changes to support them.

Welcome

ABL JDBC driver brings the power of Progress ABL to the Java world in a standardized way.

By leveraging your existing business logic the ABL JDBC driver can expose it to powerful reporting engines or data integration tools. This can help you to integrate versatile reporting solutions inside your Progress ABL application or easily respond to application and data integration requests without the pain of going to complex application changes to support them.

Features	Benefits
Standard JDBC interface	The driver implements the version 4.1 of standardized java database connectivity interface (JDBC).
Multi-tier layered architecture	Built on top of Progress Application Server the driver benefit from the proven scalability as well as other security and connectivity features like: SSL encryption and HTTP tunneling
Business logic catalog	All business logic are exposed in a meta-data catalog which offers detailed information about each registered business logic procedures and views
Meta-data support	Detailed meta-data support is available not only for the business logic but also for all connected databases: tables, columns, indexes
Multiple database support	Unlike other JDBC drivers this supports multiple connected databases, all databases connected for the Application Server shows as separate catalog – support queries against tables from multiple databases
Single entry point	The single entry point for back-end business logic facilitate security services like authentication, authorization, audit.

How can be used

While the ABL JDBC is not a end-user tool that can be used on it's own, it does provide a standard data access mechanism to your existing Progress business logic for any Java based tool that supports JDBC.

Reporting engine

While being a very powerful and productive language the Progress ABL fall short of providing a valid reporting engine that can be integrated in Progress applications. For Java there are already a great number of enterprise grade reporting engines all of which supports the JDBC database access and are able to produce pixel-perfect documents that can be viewed, printed or exported in a variety of formats; the graphical report designer available in most cases really speed-up the report design process. Among some of the most used open-source reporting engines that can be used we can name: Pentaho, Jasper Soft, Birt (Eclipse's own reporting engine).

Data integration

In today's global market place the need for application and data integration is more present than ever and continues to increase everyday. Given the embedded nature of ABL, traditionally Progress based applications offers very little integration options. This is why very often when there is a need to access data from another application (different RDBMS) or send data to it this involves implementing specific data access interfaces on one or both of the two ends. This is not only ineffective but also very difficult to maintain and expand over time to keep up with new data integration requirements. Using dedicated tools for data integration instead of application customization can offer a very scalable and easy to maintain solution while dramatically reducing the time and effort required for the data integration projects. Some of the most widely used open-source data integration tools on the Java market are: Pentaho, Talend, Apatar.

Java graphical user interface

Use ABL JDBC driver to build application graphical user interface in Java (desktop, web or smart phones), this will give you access to the Progress business logic through the common JDBC interface which will dramatically reduce the deployment effort - only one-time deployment of the JDBC driver is required, subsequent changes on the business logic does not need to be deployed as opposed to regular Open Client application deployment.

What's new

Release notes

Version 1.0 (3 Feb. 2011)

- Business logic catalog (stored procedure)
- Database Meta-Data
- Prepared and Callable Statement

Version 1.1 (16 Feb. 2011)

- Dynamic ABL Query Syntax
- BREAK-BY support with Aggregate Functions (COUNT, SUM, MIN, MAX, AVERAGE)
- Table and column alias support
- LIMIT/OFFSET add-on support (follows MYSQL implementation)

Version 1.2 (01 Aug. 2011)

- SQL basic syntax support
- Low-level data access support (CRUD)
- Improved server-side support for exception/warning (SQL Codes)
- Upgrade to version 4.1 of JDBC specification that comes with Java 7

Version 2.0 (25 Feb. 2016)

This is a major version release due to breaking changes in the API mainly following package name update but also extensions/changes that were holding back for some time.

- Dynamic procedure call support (any 4gl procedure/function)
- Improved server-side authentication service (interface API changes)
- Buffered data retrieval (pagination) for select statements and stored procedures
- Extent fields support, either whole field as array or separate extents
- Supports SSL and HTTP connection- AIA and new Pacific Application Server
- Logging service
- Documentation update

System requirements

Server-side JDBC Helper

For the server-side ABL Helper component a Progress Application Server product is required and because the implementation uses the new Object Oriented features introduced in Progress ABL language the minimum version required is 10.1C or greater, if there is any interested in making it to work with earlier versions please send us a feature request by email at contact@acorn.ro and we will take it into consideration.

For best performance the operating mode of the serving Progress Application Server should be set to 'state free' if possible, this is not a requirement "per se" as the driver can connect to an Application Server regardless of the session management operating mode used.

Client JDBC Driver

The ABL JDBC driver implements the JDBC 4.1 specification that was included in Java 7.0 therefore Java Runtime Environment (JRE) 7.0 is required.

Because the driver is using Progress© Open Client for connecting to the Application Server the Progress© Java Open Client libraries are required. The Java Open Client libraries can be found under "*java*" folder of Progress© installation (DLC) as follows:

Core Open Client Library	$DLC/java	**o4glrt.jar**
Ecore Libraries (OE V10)	$DLC/java/ext	**common.jar** **commonj.sdo.jar** **ecore.jar** **ecore.change.jar** **ecore.resources.jar** **ecore.sdo.jar** **ecore.xmi.jar**
Ecore Libraries (OE V11)	$DLC/java/ext	**common-2.2.3.jar** **ecore-2.2.3.jar** **ecore-change-2.2.3.jar** **ecore-xmi-2.2.3.jar** **tuscany-sdo-api-r2.1-1.1.1.jar** **tuscany-sdo-impl-1.1.1.jar** **tuscany-sdo-lib-1.1.1.jar** **tuscany-sdo-tools-1.1.1.jar** **xsd-2.2.3.jar**
* SSL Libraries	$DLC/java/ext	**certj.jar** **cryptoj.jar** **sslj.jar**
* SSL Certificates	$DLC/certs	**psccerts.jar**

The version of Ecore libraries changed between OE 10.2B and OE 11, make sure the right libraries are used to match the application server version.

* For SSL enabled Application Servers the SSL support libraries are required as well as the SSL certificates, if different SSL certificate is used on the application server instead of the default certificate then the custom certificate store can be specified as a connection parameter and will be used instead of the default one (psccerts.jar).

License

License

The only component that need a license is the server side of the business logic driver, the JDBC client driver has no deployment nor run-time license requirement.

The license model used is per logical computer (server/workstation) with no limitation on number of users neither named nor concurrent.

The understanding of the term logical computer - or processing environment - as used in this licensing terms is that of a physical or virtual processing environment as follows:

One Application Server serving multiple clients	One Server license
Multiple Application Servers running on the same logical computer	One Server license
Multiple logical computers (Virtual Machine) sharing the same hardware (physical computer)	One Server license per Virtual Machine

Where to Buy

The only licensed component is the server-side one, no license is required on the client and there is no restriction set on the number of concurrent/named connections.

Technology Partnership

Apart volume license discount we also offer the option of unrestricted access model for ISV/VAR under a technology partnership agreement, for more details contact us directly.

Acorn IT
48A Florilor, Floresti 407280
Cluj, Romania
Tel: +40 740 036 212
contact@acorn.ro

Evaluation

Trial

A trial version of ABL JDBC is available for an evaluation period of three months. The trial version offers the same functionality as the registered product and it does require an evaluation license to be issued.

To request a evaluation license send us an email.

Getting help

Help

You can find the most up to date version of this manual on our web-site for on-line browsing here.

Support

For each implementation we can offer tailored support and training to make sure you get everything up and running on your site and your IT staff reach a level of self-sufficiency that makes them comfortable with the product.

We also offer email support and access to defect-tracking system for clients under maintenance contracts.

We can also offer consulting services for business intelligence, data integration, enterprise data warehouse projects.

Client Data Access

This section covers the Java client-side JDBC access to the ABL business logic.

All that you need to know about how to make a connection to the Application Server, what features are supported by the JDBC driver as well as the Data Manipulation Language support and syntax.

Deployment

For database access most Java client applications uses JDBC interface, this is why the ABL JDBC driver that allows connecting to the application business logic need to be deployed on the client side, for that the 'JDBC Driver' component need to be selected during installation process.

The driver specific implementation is packed in a single jar file, however due to the fact that Progress Open Client interface is used for connecting to the Progress Application Server the driver has a dependency on Progress Open Client libraries.

In order to be able to use the JDBC driver to make the connection to the back-end Progress Application Server the driver's own JAR library as well as the libraries part of Progress Open Client need to be added to the CLASSPATH. One option will be to set the system wide CLASSPATH variable but most of the Java reporting engines and application servers does override this system default and provides other options to deploy JDBC drivers like having special library folders for JDBC drivers or having an option to adjust the application specific CLASSPATH by adding external JAR libraries – for instance in Jasper report designer (iReport) the CLASSPATH can be adjusted through Options->Classpath menu entry.

The ABL JDBC driver JAR file is named **abljdbc-x.x.x.jar** - where "x.x.x" is the product version number - and can be found in "<InstallPath>/jdbc"

For Progress Open Client the following JAR libraries need to be added to the CLASSPATH:

Core Open Client Library	$DLC/java	**o4glrt.jar**
Ecore Libraries (OE V10)	$DLC/java/ext	**common.jar** **commonj.sdo.jar** **ecore.jar** **ecore.change.jar** **ecore.resources.jar** **ecore.sdo.jar** **ecore.xmi.jar**
Ecore Libraries (OE V11)	$DLC/java/ext	**common-2.2.3.jar** **ecore-2.2.3.jar** **ecore-change-2.2.3.jar** **ecore-xmi-2.2.3.jar** **tuscany-sdo-api-r2.1-1.1.1.jar** **tuscany-sdo-impl-1.1.1.jar** **tuscany-sdo-lib-1.1.1.jar** **tuscany-sdo-tools-1.1.1.jar** **xsd-2.2.3.jar**
* SSL Libraries	$DLC/java/ext	**certj.jar** **cryptoj.jar** **sslj.jar**
* SSL Certificates	$DLC/certs	**psccerts.jar**

No Progress Open Client libraries are distributed as part of the ABL JDBC Driver installation kit, those are part of OpenEdge products and licensed by Progress Software Corporation.

Connection

From a Java client application in order to establish a connection to the data source using JDBC there are two methods that can be used:

DriverManager	A fully implemented which can connect an application to a data source, which is specified by an URL; the driver manager will take care of loading the required JDBC driver (for JDBC 4.0 drivers).
DataSource	An interface that allows details about the underlying data source to be transparent for the application (it can be registered with JNDI naming service).

Most reporting engines or data integration tools support both types of connections and ABL JDBC driver can be used with any of the two; being a JDBC 4.1 driver it does not have to be registered before using it, this is handled by the driver manager.

To make a connection using ABL JDBC driver the following information is required:

Class Name	`ro.acorn.jdbc.ABLDriver`
Connect URL	`jdbc:abl:<server>:<port>[:<name>][;<property>=value]...`

<server>	The name or address of the server that hosts the Progress Application Server to connect to.
<port>	The port number of the Progress Application Server for direct connection or the port number of the Progress Name Server on which the Application Server is registered.
<name>	Optional parameter when connecting through a Name Server the name or the Progress Application Server is required.

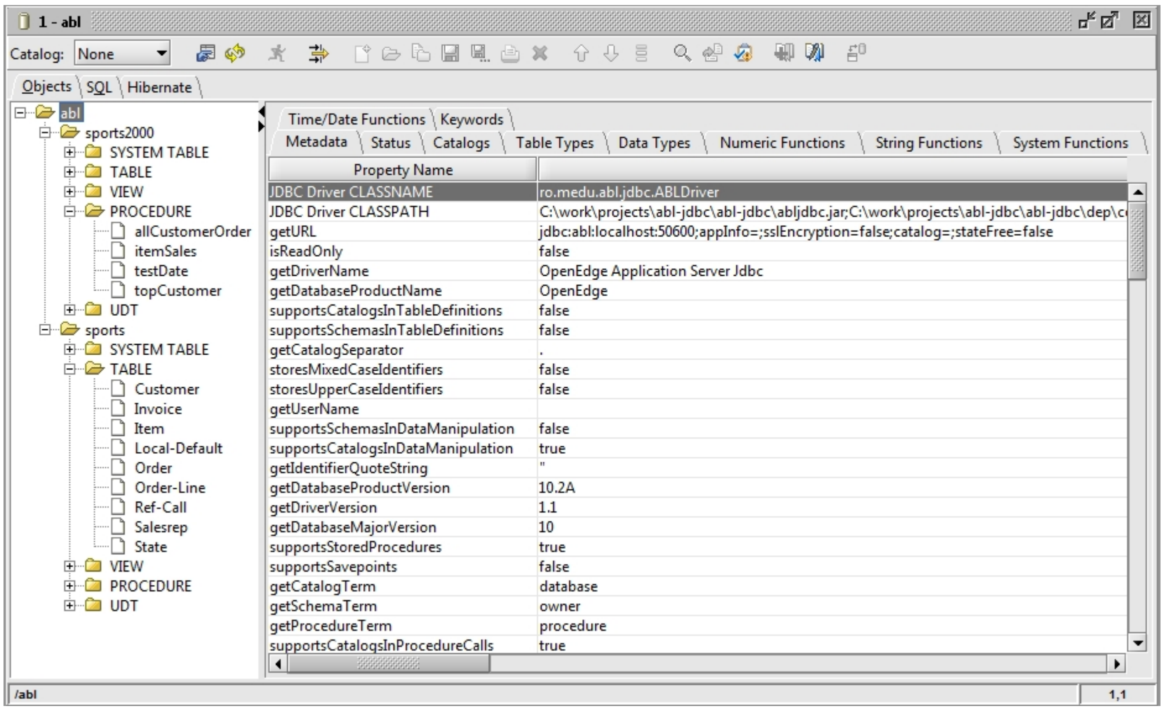
Optionally a number of driver specific properties can be specified either by setting it in the property info structure passed to connect method or directly adding them in the connection URL as pairs or property name equals value separated by semicolon.

The complete list of driver specific properties that can be set for a connection can be found in following table along with their default values.

Option	**Default**	**Description**
appInfo		Application Server connection information.
catalog		Default catalog to be used by connection.
fetchSize	500	Default statement fetch size, for large result set the records retrieval will be paged.
http	FALSE	Use HTTP protocol - AIA or new PAS.
logFile		Log file to use for the connection.
logLevel		Log level to use for current connection, valid options: ALL, TRACE, DEBUG, INFO, WARN, ERROR, FATAL
sslEncryption	FALSE	Use SSL encryption, SSL certificate support is not available for the moment.
sslStore	psccerts.jar	The store for SSL certificates.
stateFree	TRUE	Connect to a STATE-FREE Application Server, this is the recommended session management model – the driver will auto-set this property if the session management model does not match the value specified.

Database Meta Data

Database Meta Data is fully supported by the ABL JDBC driver but keep in mind that this is the 4GL database engine not the SQL one that is exposed by the Progress JDBC drivers.



Property Name	
JDBC Driver CLASSNAME	ro.medu.abl.jdbc.ABLDriver
JDBC Driver CLASSPATH	C:\work\projects\abl-jdbc\abl-jdbc\abljdbc.jar;C:\work\projects\abl-jdbc\abl-jdbc\dep\c
getURL	jdbc:abl:localhost:50600;appInfo=;sslEncryption=false;catalog=;stateFree=false
isReadOnly	false
getDriverName	OpenEdge Application Server Jdbc
getDatabaseProductName	OpenEdge
supportsCatalogsInTableDefinitions	false
supportsSchemasInTableDefinitions	false
getCatalogSeparator	.
storesMixedCaseIdentifiers	false
storesUpperCaseIdentifiers	false
getUserName	
supportsSchemasInDataManipulation	false
supportsCatalogsInDataManipulation	true
getIdentifierQuoteString	"
getDatabaseProductVersion	10.2A
getDriverVersion	1.1
getDatabaseMajorVersion	10
supportsStoredProcedures	true
supportsSavepoints	false
getCatalogTerm	database
getSchemaTerm	owner
getProcedureTerm	procedure
supportsCatalogsInProcedureCalls	true

Catalogs	Each connected database is listed as a catalog, the logical name.
Schemas	There are no multiple schemas in a Progress database, as seen by the 4GL engine, always only the "PUB" schema is listed.
Tables	All database tables are listed under corresponding catalog; detailed information for fields and indexes.
System Tables	All Progress system meta-data tables are listed.
Views	All views defined are listed on corresponding catalog but those are SQL objects that can't be accessed using regular ABL.
Procedures	Driver specific business logic services, for each catalog all procedures registered for the catalog are listed; detailed information provided for procedure parameters and returning result set(s).

Data Manipulation

Data Manipulation language supported currently by the ABL JDBC Driver is a variant of the Progress ABL language, this is because of the aim of being more easily to use by the targeted Progress ABL developers.

There are two options to access data from the underlying databases when using ABL JDBC driver:

- The Business Logic "`CALL`" statement

  This is actually the standard SQL stored procedure call statement. While Progress RDBMS does not support stored procedures in the regular 4GL database engine most of the application business logic can be exposed as a 'stored procedure' pretty much like a service in a Service Oriented Architecture.

  The Business Catalog server-side component take care of business logic stored procedures registration, it can provide meta-data information about parameters that need to be supplied as well as the result set(s) returned and it can dispatch each 'call' request to the proper business logic service.

  Using this interface means the data is not directly accessed from the underlying database(s) but through the business logic services that expose the data in a way that make most sense for the business as opposed to the somehow cryptic 3NF form internal database structure.

  The business logic call statement can return multiple result sets (dataset).

- The low level data manipulation statements (CRUD).

  This allows for direct data access to the underlying database(s) using low level CRUD data access statements using either ABL or SQL syntax.

Business Logic Call

The most important feature of the ABL JDBC driver is the possibility to abstract the database internal structure from the Java client application by exposing the application business logic as services in a way that can be seen very similar to the stored procedures available in other RDBMS. However, in this case the business logic is hosted in an application server instead of directly in the database itself to keep a strict separation between the business logic layer and the persistence layer.

The syntax used to call a business logic 'stored procedure' through ABL JDBC is.

```
{ CALL | EXEC }
   [ catalog . ] [ procedure ]
   [ ( [parameter] [ , parameter]... ) ]
```

`{ CALL | EXEC }`

While `CALL` seems the most used, `EXEC` is used by Microsoft SQL Server, any of the three keywords can be used.

`[ catalog . ] [ procedure ]`

The name of the procedure can be prepended with the catalog name, if no catalog is specified then the current catalog selected for the connection will be used. Each business logic service is registered to a catalog (database), if the procedure is not found in respective catalog a syntax error occurs.

More information about the business services available in the business catalog as well as details about parameters and returned result set(s) can be found using the database meta data functionality.

The following statement call a business logic service registered on the CRM catalog that takes no input parameters and returns all open complains that have the response due date passed.

```
CALL CRM.getComplainsWithDateDue
```

`( [parameter] [ , parameter]... )`

A business logic service can take any number of input parameters that can be mandatory or optional. The parameter's order and data type are important, if the business logic service does not validate the input parameters send a syntax error might be raised.

```
parameter: { number [ . number] | true | false | " string
" | ?}
```

Each parameter can be a numeric value - either integer or decimal, a logical value - true or false, or a quoted string. Date and date-time values can be sent as quoted string values using the ISO representation format to be correctly interpreted as a date value.

The following statement call a business logic service registered on the SPORTS catalog that takes the item number and order date as input parameters and returns all orders made on that day for selected item.

```
CALL SPORT.getItemOrders (12, "1998-11-23")
```

Dynamic Business Logic Call

While exposing business logic using the 'stored procedure' interface through the business catalog is still the recommended way because of the discovery feature of the meta-data information, sometimes it might be more convenient to call directly the 4GL procedures of the application instead of wrapping that through a 'stored procedure'.

The syntax used to call a 4GL business logic procedure through ABL JDBC is.

```
RUN
   "[ path / ]* [ procedure.p ]"
   [ ( [parameter] [ , parameter]... ) ]
```

While the `RUN` syntax was used before as an alternative to `CALL` or `EXEC` when calling 'stored procedures' the keyword is now used to directly call any 4GL procedure without having the need to create a proxy 'stored procedure' and have that registered in the business catalog.

`"[ path / ]* [ procedure.p ]"`

The 4GL external procedure name need to be available in PROPATH, relative path can be used if needed - with slash as path separator - and the full path name should be quoted.

Since there is no stored procedure defined, details about parameters and returned result set(s) are not available through the database meta data.

The following statement call a business logic procedure that takes no input parameters and returns all customers on credit hold in a single result set (temp-table).

```
RUN "crm/getCustomersOnCreditHold.p" ([out.table] ?)
```

`( [parameter] [ , parameter]... )`

A business logic service can take any number of input parameters that can be mandatory or optional. The parameter's order and data type are important, if the business logic service does not validate the input parameters sent a syntax error might be raised.

```
parameter: [direction . data type]? value
```

Because the business logic procedure is called dynamically the direction and data type of parameters sometimes is required to be specified, this is mostly true for output/input-output parameters when the data type can't be inferred from the parameter value.

```
direction: in | inout | out
```

Parameter direction simply matches the options available for passing parameters in 4GL, just using a shorter version but the mapping is self-explanatory.

```
data type: 4GL primitive data type | table | dataset
```

The data type can be any 4GL primitive data type, table for table-handle parameters and dataset for dataset-handle parameter. Since the most complex data structure returned through JDBC is the equivalent of a 4GL dataset there can either be only one output dataset parameter or one or more output table parameters.

```
value: { number [ . number] | true | false | " string " |
? }
```

Each parameter value can be numeric - either integer or decimal, logical - true or false, a quoted string or null (?). Date and date-time values can be sent as quoted string values using the ISO representation format to be correctly interpreted as a date value.

For output parameters specify null (?) as value, when using a CallableStatement question mark (?) must be used as value place-holder for every parameter and subsequently use `set` respectively `registerOutputParameter` methods of the CallableStatement.

The following statement call a business logic procedure that takes the item number and order date as input parameters and returns all orders made on that day for selected item including the corresponding order lines in a single dataset.

```
RUN getItemOrders (12, "1998-11-23", [out.dataset] ?)
```

ABL

Low level ABL Data Manipulation Language statements supports all CRUD (create, read, update, delete) operations against connected database(s) using a syntax very close to the ABL equivalent in order to be more comfortable for developers that use the Progress ABL as the main development language.

The ABL select statement is supporting almost all options available on a native ABL static FOR EACH statement - including break-by and accumulate functions - and add-up some functionality like limiting the number of records retrieved and option to specify a starting offset.

For Each

The syntax supported by the ABL JDBC driver for dynamic data retrieval directly from the underlying database(s) is an extended version of plain Progress© ABL FOR statement.

```
FOR EACH recordPhrase
   [ , { EACH | FIRST | LAST } recordPhrase ]...
   [ BREAK ]
   [ BY orderField [ DESCENDING ] ]...
   [ LIMIT { offset, limit | limit [ OFFSET
offset ] } ]
```

EACH

Retrieve all records from the table that meets the criteria, this is mandatory for the first record phrase specified for the select statement. Achieving the same functionality that FIRST or LAST options provides can be done by using LIMIT 1 option accompanied by the DESCENDING order-by option (for LAST).

For instance to select the first customer record by customer name the following statement can be used.

```
FOR EACH Customer BY Name LIMIT 1
```

FIRST

Retrieve only the first record from the table that meets the criteria, this can be used for subsequent record phrases specified for the select statement.

```
FOR EACH Customer, FIRST Order OF Customer
```

LAST

Retrieve only the last record from the table that meets the criteria, this can be used for subsequent record phrases specified for the select statement.

The following statement retrieves all customers and the last order made by each of them, note that customers that do not have orders are exempted from the result set in this case (see OUTER-JOIN options).

```
FOR EACH Customer, LAST Order OF Customer BY OrderDate
```

`[ EACH | FIRST | LAST ] recordPhrase`

Specifies the tables to retrieve data from, EACH option must be used for the first table while for the subsequent tables specified any of the three options are valid.

```
table [ AS alias ]
   [ FIELDS ( * | { field | aggregate}... )
     | EXCEPT ( field... ) ]
   [ [ LEFT ] OUTER-JOIN ]
   [ OF parent ]
   [ WHERE criteria ]
   [ USE-INDEX index ]
```

`table [ AS alias ]`

The name of the table from which the records are to be retrieved, this can be prepended with the database 'catalog' name if exists in more than one database; queries across tables from multiple databases are supported as in regular ABL.

Optionally the table name can be aliased with a shorter name to save typing very long names in subsequent where clause or when query need to be made on a hierarchical self-referenced table; if an alias is specified for a given table then the alias must be used instead of the table name in WHERE criteria clause as well as in BY order options. The alias names used must be unique for the statement.

The following statement retrieves all customer's complaints where the customer retention 'silo' has a separate database.

```
FOR EACH Sports.Customer AS c,
    EACH Crm.Complaint AS d
    WHERE d.CustNum EQ c.CustNum
```

The following statement retrieves all follow-up calls from the self-referencing RefCall table including the date and message of the referenced call.

```
FOR EACH RefCall AS c,
   EACH RefCall AS p FIELDS (CallDate AS ParentDate, Txt AS
ParentTxt)
   WHERE p.CallNum EQ c.CallNum
```

```
FIELDS ( * | { { field | field[extent] | aggregate} [ AS
alias ]} [ , ... ] )
```

The FIELDS option specifies which fields to be retrieved from each table used in select statement, if no fields are specified then all fields from the table are retrieved in the result set - specifying '*' (asterisk) as field selection also selects all fields from the table.

Extent fields can be either selected as an array (please note not all SQL clients might be able to handle the `java.sql.Array` fields) when all you have to specify is the field name, or as individual extents using the square bracket notation.

For group-by select statements that use the BREAK option the field list specified for the FIELDS option can take an aggregate function attached. The aggregate functions available are: COUNT, SUM (TOTAL), AVERAGE (AVG), MINIMUM (MIN), MAXIMUM (MAX).

For all fields used in FIELDS phrase an alias can be used to make the column name more readable in the result set, also this can be used when fields with the same name from different tables are required since only one column with a given name can exist in the result set.

As opposed to the table name aliasing where the alias need to be used for referencing the table in subsequent WHERE clause and BY order list, the column alias can not be used instead of the column name for referencing it in WHERE clause or BY order list.

The following statement retrieves the number of orders and most recent order date grouped by order status.

```
FOR EACH Order
   FIELDS (OrderStatus, COUNT(*) AS Orders, MAX(OrderDate) AS
LastOrder)
   BREAK BY OrderStatus
```

The following statement can be used to get the order status as well as the order line status when the field name is the same in both tables.

```
FOR EACH Order FIELDS (Status AS OrderStatus),
   EACH OrderLine FIELDS (Status AS LineStatus) OF Order
```

Note: Specifying fields that do not exist in the table for the FIELD list is considered a syntax error and no result set is returned - an SQLException is thrown; this applies whether an aggregate function is specified for the field or not.

When no FIELDS list is specified for a table then all fields from that table that were already added to the result set from previous tables are simply ignored and will be missing from the result set, this is not considered an error.

```
EXCEPT ( field [ , ... ] )
```

The EXCEPT option can be used when all but a small number of fields are to be retrieved from the table, as an alternative to specify a longer field selection list using FIELDS option.

The following statement retrieves all fields from the order tables except the customer number and the instructions as well as the customer name which is aliased to 'Customer'.

```
FOR EACH Customer FIELDS (Name as Customer),
   EACH Order EXCEPT (CustNum, Instructions) OF Customer
```

Note: Specifying fields that do not exist in the table for the EXCEPT list is not considered a syntax error.

```
[ LEFT ] OUTER-JOIN
```

Specify a outer join between the current table and one of the tables specified in previously record phrases of the select statement. See your current Progress documentation for more information and/or specific notes.

```
OF parent
```

Specify a implicit join between the current table and one of the tables specified in previously record phrases of the select statement, meaning there is no need to specify the join using a regular WHERE clause but let the Progress AVM infer that join clause. See your current Progress documentation for more information and/or specific notes.

If the referenced table was aliased then the alias should be used instead of the table name.

```
FOR EACH Customer AS c,
   EACH Order OUTER-JOIN OF c
```

`WHERE criteria`

Specify the selection criteria used while selecting records. Although all selection criteria can be set in the record phrase of the last table used in select it is recommended to specify table specific selection criteria in the table's own record phrase for performance considerations.

`USE-INDEX index`

Specify explicitly the index to be used while selecting records. See your current Progress documentation for more information and/or specific notes.

The following statement will retrieve all customer records ordered by sales representative.

```
FOR EACH Customer USE-INDEX SalesRep
```

Note: Although functionally equivalent as specifying a number of order BY clauses the fields of the index specified by USE-INDEX option will not be considered by the BREAK option.

`BREAK`

Can only be used if at least one BY option is specified and the result will be that the result set will only contain a single record for the group selection, this is functionally equivalent to the SQL "GROUP BY" statement.

When used no other fields beside those listed in the BY list can be specified in the FIELDS list unless having an aggregate function attached.

This is often used together with aggregate functions on field selections to obtain aggregated field values over a group of records but it can also be used to get distinct group values.

The following statement retrieves the total number of customers per country together with the average account balance.

```
FOR EACH Customer FIELDS(Country, COUNT(*), AVG(Balance)) BREAK
BY Country
```

Note: Although using the same syntax as in regular ABL open query statement the BREAK option has a somehow different functionality adapted to the result-set based data retrieval of the dynamic select statement.

`BY orderField [ DESCENDING ]`

Specify the order in which the records are to be returned, the DESCENDING option sorts the records in descending order. The `orderField` expression can be just the field name if the field can be unique identified for the tables used in select clause or it can be pre-pended with the table name or alias if specified, see the record phrase section for more details about table aliasing.

`LIMIT {offset, limit | limit OFFSET offset}`

The LIMIT clause can be used to constrain the number of records retrieved by the select statement. The LIMIT option can take one or two numeric arguments both of which needs to be non-negative integer constants.

The one argument only syntax let you limit the number of rows returned starting with the first record that meets selection criteria.

The following statement will return all customers and their orders but limit the number of records in returning result set to ten; depending on the data in the tables those records can be orders of only one customer or more, it does not returns all orders for the first ten customers.

```
FOR EACH Customer, EACH Order OF Customer LIMIT 10
```

For the two arguments option there are two different syntax format that are allowed both of them does the same thing by limiting the number of records returned in the result set but starting from the 'offset' record instead of first record that meets the select criteria. The offset is a zero-based index value meaning that the offset of the first record is 0 (not 1), if the offset is greater than the number of records that meets the selection criteria no record is returned.

The following statements are functionally equivalent, both returns ten records starting from the fifth position - because the offset is zero based this actually means the result set will include records from 6 to 15.

```
FOR EACH Customer, EACH Order OF Customer LIMIT 5, 10

FOR EACH Customer, EACH Order OF Customer LIMIT 10 OFFSET 5
```

To return all records starting from a given offset position specify zero for the LIMIT value, the following statement will return all records that meet select criteria starting from fifth position to the end.

```
FOR EACH Customer, EACH Order OF Customer LIMIT 5, 0

FOR EACH Customer, EACH Order OF Customer LIMIT 0 OFFSET 5
```

Create

The syntax supported by the ABL JDBC driver for inserting new records directly into the underlying database(s) is using a combination between regular Progress© ABL CREATE and ASSIGN statements.

```
CREATE tableName
   ASSIGN fieldName = value [ [ , ] fieldName = value
]...
```

The following statement creates a new record into the State table.

```
CREATE State
   ASSIGN State = 'CJ',
          Statename = 'Cluj',
          Region = 'NW'
```

Notes:

- Automatic data-type conversion does occur if possible - setting a character field to a numeric value will result in field having the string representation of the value. If data-type conversion fails the transaction is rolled-back and an exception is raised.
- If no CREATE privilege exists on the table or unique constraints is violated the transaction is rolled-back and an exception is raised.
- CREATE/WRITE triggers does fire as for regular ABL.
- No functions are supported in field assignment, only scalar values.
- The comma between fields assignment is optional but used for 'readability'.

Update

The syntax supported by the ABL JDBC driver for updating existing records directly into the underlying database(s) is using a combination between SQL UPDATE and regular Progress© ABL ASSIGN statements.

```
UPDATE tableName
   ASSIGN fieldName = value [ [ , ] fieldName = value
]...
   [ WHERE criteria ]
```

Because of the result-set nature of SQL UPDATE and the sequential record-processing of the Progress ABL the functionality is implemented by simply using ASSIGN statement inside of data retrieval block (FOR EACH) - records are updated one by one inside the retrieval block.

The following statement updates the department code of all employees from Marketing department (code '300') and transfer them to the Sales department (code '400').

```
UPDATE Employee
   ASSIGN DeptCode = '400'
   WHERE DeptCode = '300'
```

Notes:
- Automatic data-type conversion does occur if possible - setting a character field to a numeric value will result in field having the string representation of the value. If data-type conversion fails the transaction is rolled-back and an exception is raised.
- If no WRITE privilege exists on the table or unique constraints is violated the transaction is rolled-back and an exception is raised.
- If a record that meets the selection criteria is locked by some other user the transaction is rolled-back and an exception is raised. If selection criteria leads to a full table scan (WHOLE-INDEX) the record lock is only attempted for the records that meets the selection criteria.
- WRITE triggers does fire as for regular ABL.
- No functions are supported in field assignment, only scalar values.
- The comma between fields assignment is optional but used for 'readability'.

Delete

```
DELETE tableName
   [ WHERE criteria ]
```

Because of the result-set nature of SQL DELETE and the sequential record-processing of the Progress ABL the functionality is implemented by simply using ABL DELETE statement inside of data retrieval block (FOR EACH) - records are deleted one by one inside the retrieval block.

The following statement deletes all employees from Marketing department (code '300').

```
DELETE Employee
   WHERE DeptCode = '300'
```

Notes:

- If no DELETE privilege exists on the table or validation expression evaluate to false the transaction is rolled-back and an exception is raised.
- If a record that meets the selection criteria is locked by some other user the transaction is rolled-back and an exception is raised. If selection criteria leads to a full table scan (WHOLE-INDEX) the record lock is only attempted for the records that meets the selection criteria.
- DELETE triggers does fire as for regular ABL.

SQL

Low level SQL Data Manipulation Language statements supports all CRUD (create, read, update, delete) operations against connected database(s) using a basic SQL syntax to accommodate developers more comfortable with the standard SQL language.

Even if the SQL language is used in that case the data access is still done through the ABL database engine therefore some notable facts need to be considered:

- not all SQL functionality is implemented (some of the most notable ones that are missing for the moment are: the IN construct in SELECT statement, functions support in field assignments)
- transactions can't span multiple statements, if more statements need to be part of the same transaction those can be executed together as a batch through a single call
- table triggers and validation expression does apply (same as for regular ABL statements)
- for select statements no cost-based optimizer is used but the regular run-time index selection same as for ABL dynamic queries - update statistics is not required nor does affect queries performance in any way.

Select

The SQL syntax supported by the ABL JDBC driver for dynamic data retrieval directly from the underlying database(s) is a limited implementation of SQL-92 syntax.

```
SELECT fieldPhrase
   FROM tablePhrase [ , tablePhrase ]...
  [ WHERE criteria ]
  [ BREAK ]
  [ BY orderField [ DESC | DESCENDING ] ]...
  [ LIMIT { offset, limit | limit [ OFFSET
offset ] } ]
```

fieldPhrase

```
 * | { { field | field[extent] | aggregate} [ AS
alias ]} [ , ... ]
```

The field phrase specifies which fields to be retrieved from tables used in select statement. Field phrase is mandatory for SQL statement - specifying '*' (asterisk) as field selection selects all fields from all tables used in select.

Specifying the table name is mandatory for each field of the field phrase - exception is allowed only when there is a single table used in select; if the table name was aliased then the table alias can be used instead.

Selecting all fields from one of the tables used in SELECT statement can be accomplish by simply specifying 'table.*' in the field selection phrase.

Extent fields can be either selected as an array (please note not all SQL clients might be able to handle the `java.sql.Array` fields) when all you have to specify is the field name, or as individual extents using the square bracket notation.

The following statement retrieves all order information and the customer name of all orders.

```
SELECT o.*, c.Name AS Customer
   FROM Order AS o, Customer AS c
   WHERE c.CustNum = o.CustNum
```

For group-by select statements that use the BREAK option the fields list specified in the field phrase can take an aggregate function attached. The aggregate functions available are: COUNT, SUM (TOTAL), AVERAGE (AVG), MINIMUM (MIN), MAXIMUM (MAX).

For both plain fields or aggregates used in field phrase an alias can be used to make the column name more readable in the result set, also this can be used when fields with the same name from different tables are required since only one column with a given name can exist in the result set.

As opposed to the table name aliasing where the alias need to be used for referencing the table in subsequent WHERE clause and BY order list, the column alias can not be used instead of the column name for referencing it in WHERE clause or BY order list.

The following statement retrieves the number of orders and most recent order date grouped by order status.

```
SELECT o.OrderStatus, COUNT(*) AS Orders,
   MAX(o.OrderDate) AS LastOrder
   FROM Order AS o
    BREAK BY o.OrderStatus
```

The following statement can be used to get the order status as well as the order line status when the field name is the same in both tables.

```
SELECT o.Status AS OrderStatus, l.Status AS LineStatus
   FROM Order AS o, OrderLine AS l
    WHERE l.OrderNum = o.OrderNum
```

Note: Specifying fields that do not exist in the table for the field phrase is considered a syntax error and no result set is returned - an SQLException is thrown; this applies whether an aggregate function is specified for the field or not.

`tablePhrase`

```
table [ AS alias ]
   [ [ INNER | LEFT | RIGHT] JOIN parentTable ON
criteria ]
```

Specifies the tables to retrieve data from with optional joining criteria.

`table [ AS alias ]`

The name of the table from which the records are to be retrieved, this can be prepended with the database 'catalog' name if exists in more than one database; queries across tables from multiple databases are supported as in regular ABL.

Optionally the table name can be aliased with a shorter name to save typing very long names in subsequent where clause or when query need to be made on a hierarchical self-referenced table; if an alias is specified for a given table then the alias must be used instead of the table name in WHERE criteria clause as well as in BY order options. The alias names used must be unique for the statement.

The following statement retrieves all customer's complaints where the customer retention 'silo' has a separate database.

```
SELECT *
   FROM Sports.Customer AS c
   JOIN Crm.Complaint AS d
   ON d.CustNum = c.CustNum
```

The following statement retrieves all follow-up calls from the self-referencing RefCall table including the date and message of the referenced call.

```
SELECT c.*, p.CallDate AS ParentDate, p.Txt AS ParentTxt
   FROM RefCall AS c, RefCall AS p
   WHERE p.CallNum = c.CallNum
```

`[ INNER | LEFT | RIGHT ] JOIN parentTable ON criteria`

Specify a join between the current table and the table specified in previous record phrase of the select statement. If not specified the join mode is implied as being inner join, full outer join is not supported.

Normally criteria specified on table join is a series of equality conditions between fields that define the relation between the two tables, however for performance considerations additional filter conditions on the joined table (child table) can also be specified on the join criteria instead of being added in the global where clause.

The following statement retrieves all customer's orders filtering only those of a particular sales representative.

```
SELECT *
   FROM Customer AS c
   JOIN Order AS o
   ON o.CustNum = c.CustNum AND o.SalesRep = "HXM"
```

`WHERE criteria`

Specify the selection criteria used while selecting records.

`BREAK`

Can only be used if at least one BY option is specified and the result will be that the result set will only contain a single record for the group selection, this is functionally equivalent to the SQL "GROUP BY" statement.

When used no other fields beside those listed in the BY list can be specified in the FIELDS list unless having an aggregate function attached.

This is often used together with aggregate functions on field selections to obtain aggregated field values over a group of records but it can also be used to get distinct group values.

The following statement retrieves the total number of customers per country together with the average account balance.

```
SELECT Country, COUNT(*), AVG(Balance) FROM Customer BREAK BY
Country
```

`BY orderField [ DESC | DESCENDING ]`

Specify the order in which the records are to be returned, the DESCENDING option sorts the records in descending order. The `orderField` expression can be just the field name if the field can be unique identified for the tables used in select clause or it can be pre-pended with the table name or alias if specified, see the `record phrase` section for more details about table aliasing.

`LIMIT {offset, limit | limit OFFSET offset}`

The LIMIT clause can be used to constrain the number of records retrieved by the select statement. The LIMIT option can take one or two numeric arguments both of which needs to be non-negative integer constants.

The one argument only syntax let you limit the number of rows returned starting with the first record that meets selection criteria.

The following statement will return all customers and their orders but limit the number of records in returning result set to ten; depending on the data in the tables those records can be orders of only one customer or more, it does not returns all orders for the first ten customers.

```
SELECT * FROM Customer AS c, Order AS o WHERE o.CustNum =
c.CustNum  LIMIT 10
```

For the two arguments option there are two different syntax format that are allowed both of them does the same thing by limiting the number of records returned in the result set but starting from the 'offset' record instead of first record that meets the select criteria. The offset is a zero-based index value meaning that the offset of the first record is 0 (not 1), if the offset is greater than the number of records that meets the selection criteria no record is returned.

The following statements are functionally equivalent, both returns ten records starting from the fifth position - because the offset is zero based this actually means the result set will include records from 6 to 15.

```
SELECT * FROM Customer AS c, Order AS o
   WHERE o.CustNum = c.CustNum  LIMIT 5, 10

SELECT * FROM Customer AS c, Order AS o
   WHERE o.CustNum = c.CustNum  LIMIT 10 OFFSET 5
```

To return all records starting from a given offset position specify zero for the LIMIT value, the following statement will return all records that meet select criteria starting from fifth position to the end.

```
SELECT * FROM Customer AS c, Order AS o
   WHERE o.CustNum = c.CustNum  LIMIT 5, 0

SELECT * FROM Customer AS c, Order AS o
   WHERE o.CustNum = c.CustNum  LIMIT 0 OFFSET 5
```

Insert

```
INSERT INTO tableName
   [ ( fieldName [ , fieldName ]... ) ]
   VALUES ( value [ , value ]... )
```

The following statement creates a new record into the State table:

```
INSERT INTO State
   (State, StateName, Region)
   VALUES ('CJ', 'Cluj', 'NW')
```

or specify only the field values in the proper order:

```
INSERT INTO State
   VALUES ('CJ', 'Cluj', 'NW')
```

Notes:

- Automatic data-type conversion does occur if possible - setting a character field to a numeric value will result in field having the string representation of the value. If data-type conversion fails the transaction is rolled-back and an exception is raised.
- If no CREATE privilege exists on the table or unique constraints is violated the transaction is rolled-back and an exception is raised.
- CREATE/WRITE triggers does fire as for regular ABL.
- No functions are supported in field assignment, only scalar values.

- If fields name list is omitted and only values list is specified the number and the order of values entries must match exactly the number and order of fields in the table or an exception is raised.

Update

Because of the result-set nature of SQL UPDATE and the sequential record-processing of the Progress ABL the functionality is implemented by simply using ASSIGN statement inside of data retrieval block (FOR EACH) - records are updated one by one inside the retrieval block.

```
UPDATE tableName
   SET fieldName = value [ [ , ] fieldName =
value ]...
   [ WHERE criteria ]
```

The following statement updates the department code of all employees from Marketing department (code '300') and transfer them to the Sales department (code '400').

```
UPDATE Employee
   SET DeptCode = '400'
   WHERE DeptCode = '300'
```

Notes:

- Automatic data-type conversion does occur if possible - setting a character field to a numeric value will result in field having the string representation of the value. If data-type conversion fails the transaction is rolled-back and an exception is raised.
- If no WRITE privilege exists on the table or unique constraints is violated the transaction is rolled-back and an exception is raised.
- If a record that meets the selection criteria is locked by some other user the transaction is rolled-back and an exception is raised. If selection criteria leads to a full table scan (WHOLE-INDEX) the record lock is only attempted for the records that meets the selection criteria.
- WRITE triggers does fire as for regular ABL.
- No functions are supported in field assignment, only scalar values.
- The comma between fields assignment is optional but used for 'readability'.

Delete

```
DELETE FROM tableName
   [ WHERE criteria ]
```

Because of the result-set nature of SQL DELETE and the sequential record-processing of the Progress ABL the functionality is implemented by simply using ABL DELETE statement inside of data retrieval block (FOR EACH) - records are deleted one by one inside the retrieval block.

The following statement deletes all employees from Marketing department (code '300').

```
DELETE Employee
   WHERE DeptCode = '300'
```

Notes:

- If no DELETE privilege exists on the table or validation expression evaluate to false the transaction is rolled-back and an exception is raised.
- If a record that meets the selection criteria is locked by some other user the transaction is rolled-back and an exception is raised. If selection criteria leads to a full table scan (WHOLE-INDEX) the record lock is only attempted for the records that meets the selection criteria.
- DELETE triggers does fire as for regular ABL.

Business Logic

This section covers the Progress ABL server-side component that is used by the JDBC driver and which provides a number of generic or JDBC specific functionalities:

Database Meta Data
Business Catalog
Dynamic Select Engine
Core Services

Deployment

The server side component of the ABL JDBC driver is a collection of classes implemented in Progress ABL that serves as a façade for the application business logic and need to be made available on the Progress Application Server that hosts the application itself.

The only thing that need to be in order to have it deployed is to add the ABL JDBC server folder to the agent's PROPATH for the Progress Application Server used by the application, there are no extra databases required or anything more that that.

The ABL JDBC server component can be found in "<InstallPath>/abl" folder, for that the "ABL Server" component need to be selected during installation process.

License
The license file need to be named **abljdbc.lic** (previously abljdbcsrv.lic) and placed in the same folder where the ABL JDBC server component was installed - "<InstallPath>/abl". The license validation process will also search for the license file through the Progress Application Server PROPATH.

Controller

`ro.acorn.jdbc.sql.Controller`

The central piece of the business logic framework is the 'controller' which will boot-strap all framework's services as needed and serves as the interface between the client API requests and the application business logic by automatically providing session management, authentication and authorisation for each and every request.

static IAuthentication getAuthenticationService ()
Static method that gives access to the authentication service used by the framework, if authentication is not enabled the AnonymousAuthentication service is used.

static IAuthorization getAuthorizationService ()
Static method that gives access to the authorization service used by the framework, if any.

static ICatalog getCatalogService ()
Static method that gives access to the business catalog service used by the framework.

static ClientPrincipal getClientInfo ()
Returns a ClientPrincipal instance holding information about the current user - even if authentication is not required this will contain information about user session like the login name provided and session start date. Stored procedures and business views can use this information to restrict information to sensitive data (row level data filtering) or provide custom user's view on the business data.

static IConfiguration getConfigurationService ()
Static method that gives access to the configuration service used by the framework.

static ILogger getLogger ()
Static method that gives access to the logging service used by the framework, if any.

Stored Procedure

ro.acorn.jdbc.sql.IStoredProcedure

In order to be able to expose the business logic as a service this simple interface need to be implemented by each class that is going to be registered in the business logic catalog. If your business logic supports pagination then is better to also implement the IBufferedProcedure interface.

ProcedureMetaData getMetaData

The business logic service need to be able to self-describe itself in order for the business catalog registration service to register it.

The method should return a valid instance of ProcedureMetaData object that can be obtained by simply calling the constructor providing the procedure name which is mandatory and optionally the description of the business logic service as it's going to appear on business catalog. If the stored procedure does not return a valid instance of procedure meta data or if the service name meta data information was not set then the procedure fails to be registered into the business catalog.

If the business logic service accepts parameters (input, input-output or output), returns a value or a result set there are methods in ProcedureMetaData object to describe those parameters as well. Although this is not mandatory it can provide more details on how the business logic service (stored procedure) need to be called and what the result looks like.

```
integer executeProcedure
    (callStatement as CallableStatement,
     output resultSet as handle)
```

This method is called when a store procedure call statement is made from the JDBC client; normally this should call some service of existing application business logic and depending on input parameters received it can set output parameters and return a primitive data type, a temp-table or dataset handle.

The method have two parameters, first is an instance of CallableStatement that can be used to retrieve the input parameters sent from JDBC client and set values of output parameters; last parameter of the method is an output handle to a temp-table or a dataset containing the result set(s).

Since the output parameter can be either a temp-table or dataset a simple handle is used instead of table-handle or dataset-handle, this means that the data structure must still be valid (not be deleted if dynamically created) when the method ends. It is expected however that, if dynamic, the result data structure to be deleted by the stored procedure's destructor in order to avoid memory leaks.

Standard JDBC does allows for multiple result sets and the ABL JDBC driver supports multiple open result sets.

Procedure Meta Data

ro.acorn.jdbc.sql.metadata.ProcedureMetaData

This is a way for a business logic service that is going to be registered in the business logic catalog to describe itself by providing some useful meta-data information like service description as well as parameter information.

The kind of possible procedure return types are defined as static constants:

`procedureNoResult`	the procedure does not return any result
`procedureReturnsResult`	the procedure returns a result
`procedureResultUnknown`	the procedure result is unknown

ProcedureMetaData
(serviceName as character,
serviceDescription as character)

Public constructor which creates a meta data object for the given service name having an optional service description; the return type unknown by default. The service name need to be unique for the catalog in order to be able to register it in the business catalog.

character getName ()
Returns the business service name as is going to be shown in business catalog.

character getDescription ()
Returns the business service description as is going to be shown in business catalog.

integer getReturnType ()
Returns the business service return type, if no return parameter was set and the return type was not set to no return it will be return unknown by default.

integer getParameterNumber ()
Returns the number of parameters the business service takes, those can be input, output or input-output.

ParameterMetaData getParameter (parameterIndex as integer)
Returns the meta data information for specific business service

parameter, the parameter index is one-based. If parameter index specified is less than one or greater than the number of parameters null is returned.

ParameterMetaData getReturnParameter ()
Returns the meta data information for return parameter if the business service returns any. If no return parameter was registered for the business service null is returned.

```
void addParameter (name as character, type as integer,
          dataType as character, description as character,
          nullable as integer)
```
Add a new parameter to the business service parameter list, the parameters should be added in order as it is impossible to alter the order once registered.

- name is mandatory and unique
- type can be either input, output or input-output
- data type is the Progress data type (translated to SQL equivalent by the DatabaseMetaData)
- description is optional but is preferable to be filled in
- nullable can be either set to nullable, non nullable or unknown

Parameter types as well as nullable options are defined as static constants in ParameterMetaData.

void setName (name as character)
Sets the business service name as is going to be shown in business catalog.

void setDescription (description as character)
Sets the business service description as is going to be shown in business catalog.

void setReturnType (returnType as integer)
Sets the business service return type, valid options are: `procedureNoResult` and `procedureResultUnknown`. The return type is set to `procedureReturnsResult` only if the return parameter is set by calling `setReturn` method.

```
void setReturn (name as character, dataType as character,
               description as character)
```
Sets the return parameter of the business service, there can only be one return parameter. Use this method when the return parameter is a scalar one.

- name is mandatory and unique
- data type is the Progress data type (translated to SQL equivalent by the DatabaseMetaData)
- description is optional but is preferable to be filled in

When set the procedure return type is changed to `procedureReturnsResult.`

void setReturn (name as character, description as

```
character,
                    resultSet as handle)
```

Sets the return parameter of the business service, there can only be one return parameter. Use this method when the return parameter is a result set; if the resultSet handle is a dataset then the business service returns multiple result sets which is possible as opposed with only one scalar return parameter.

- name is mandatory and unique
- description is optional but is preferable to be filled in
- resultSet is mandatory and must be a valid-handle to either a temp-table or a data-set object.

When set the procedure return type is changed to `procedureReturnsResult`.

Parameter Meta Data

ro.acorn.jdbc.sql.metadata.ParameterMetaData

Used to describe each business services parameters in order to have the complete detail of the parameters a business service expects.

The kind of possible parameter types are defined as static constants:

`procedureColumnUnknown`	unknown parameter type
`procedureColumnIn`	input parameter
`procedureColumnInOut`	input-output parameter
`procedureColumnOut`	output parameter
`procedureColumnReturn`	return parameter
`procedureColumnResult`	return result set parameter

The parameter nullable options are defined as static constants:

`columnNullableUnknown`	nullable option is unknown
`columnNoNulls`	parameter does not allow null value
`columnNullable`	parameter does allow null value

```
ParameterMetaData
     (name as character, type as integer
     dataType as character, description as character,
     nullable as integer)
```

Public constructor which creates a meta data object for a scalar parameter.

- name is mandatory
- type can be either input, output, input-output or return
- data type is the Progress data type (translated to SQL

equivalent by the DatabaseMetaData)
- description is optional but is preferable to be filled in
- nullable can be either set to nullable, non nullable or unknown

```
ParameterMetaData
     (name as character, description as character,
     resultSet as handle)
```

Public constructor which creates a meta data object for a result-set return parameter.

- name is mandatory
- description is optional but is preferable to be filled in
- resultSet must be a valid-handle of either a temp-table or a data-set object

The type of parameter is automatically set to `procedureColumnResult`.

Getter and setter methods are available for each parameter attribute.

Buffered Procedure

`ro.acorn.jdbc.sql.IBufferedProcedure`

The result sets can sometime be very large retrieving all data at once in a temp-table or dataset structure might not be the best approach, sometimes even impossible as temp-table storage buffers are overflow. To overcome this the stored procedure might choose to implement the IBufferedProcedure interface, after the initial call to `executeProcedure` which will retrieve the first data page the client will callback for more data using `fetchPage` method.

The pagination is controlled by the buffered procedure alone, the page size requested by the client is only an indication unless it's set to be zero which means no pagination is to be used and the client expects to receive all data at once. The procedure might choose to set hard limit on maximum number of records retrieved in a data page but if the client request all records (page size set to zero) it will not ask for more data even if the business logic returns less than the total number of records. It is recommended to better throw an error if number of records that are to be returned exceeds the hard limit instead of sending incomplete result set, that way the client is at least aware of the problem and might try to retrieve the records using a lower page size.

`void setFetchSize (pageSize as integer)`
Set the page size when fetching data, this is set by the client but the stored procedure might choose to consider this just as an indication - the client will use all records returned regardless of the fetch size.

`logical hasMoreData ()`
Returns true if there is more data to be retrieved.

`logical fetchPage (output resultSet as handle)`
Fetch new data page and return true if there is more data available or false if all data was retrieved.
Same as with `executeProcedure` the output parameter can be either a temp-table or dataset handle but it must have the same data structure as the initial result returned by `executeProcedure`.
Because the client does not actually control pagination, the current offset is to be kept by the business procedure itself - the procedure is instantiated only once and the controller will subsequently call `fetchPage` on the same instance as requested by the client either till all data is retrieved or the client doesn't need more data (closes the result set).

Statements

Each request sent from the JDBC client is a 'statement', depending on the syntax used on that statement it can be one of the three:

Callable Statement	store procedure call
Select Statement	dynamic select
Batch Statement	batch of data manipulation statements that do not return result set

Batch and dynamic select statements are normally handled internally by the framework, the callable statement is passed to the business logic service on each call.

For all types of statements there is an option to raise error condition(s) that are going to be sent back to the client and a SQLWarning exception is going to be thrown by the driver.

For business logic services this can be used to validate input parameters or signal abnormal execution that made impossible to return the result set expected by the client, if the business service normally returns a result set and fails to provide one an error condition need to be raised - if no error condition is set from the business logic and no valid result set is returned a generic exception is raised by the driver.

Statement

`ro.acorn.jdbc.sql.Statement`

When a business logic service is called from the JDBC client the stored procedure class that implements the service receive an instance of CallableStatement that holds all input parameters that were set by the client and which can also be used to set output/ return parameter values back to the client - the client need to use a callable statement for that and register the output parameters before making the call.

This object extends the base statement object by adding functionality to retrieve and set parameters that can be passed between client and server and the other way around.

The data type of each parameter sent is important to match the data type expected for that parameter, parameters are retrieved using index position which is a one-base index (starting from 1). The callable statement provides methods to get or set parameters for various data types.

`void addException (errorObj as Progress.Lang.Error)`
Add a new error condition(s) by retrieving all messages from the error object using the SQLState "generic error".

```
void addException (errorObj as Progress.Lang.Error,
                   sqlState as character)
```
Add a new error condition(s) by retrieving all messages from the error object using the given SQLState code.

`void addException (sqlErr as SQLException)`
Add a new error exception, this can have a full chain of cause exceptions that tracks back to the root one.

```
void addWarning (reason as character, code as integer,
                 sqlState as character)
```
Add a new warning message specifying the reason, the native code and the SQLState code (see values available in `ro.medu.abl.sql.SQLState`).

`void addWarning (reason as character, code as integer)`
Add a new warning message specifying the reason and native code, the SQLState is going to be set to "generic error".

`void addWarning (reason as character)`
Add a new warning message specifying only the reason, the native code is zero and the SQLState is "generic error".

```
void addWarning (errorObj as Progress.Lang.Error,
                 sqlState as character)
```
Add a new warning message(s) by retrieving all messages from the error object and use given SQLState code - GetMessage and GetMessageNum methods of error object are used for reason and error code.

`void addWarning (errorObj as Progress.Lang.Error)`
Add a new warning message(s) by retrieving all messages from the error object using the SQLState "generic error".

`void addWarning (warnObj as SQLWarning)`
Add a new warning message(s) by retrieving all messages from the

SQLWarning object with corresponding SQLState code set for each error message.

void clearWarnings ()
Remove all SQL warnings set on this statement.

ClientPrincipal getClientInfo ()
Returns a ClientPrincipal instance holding information about the current user - even if authentication is not required this will contain information about user session like the login name provided and session start date. Stored procedures can use this information to restrict information to sensitive data (row level data filtering) or provide custom user's view on the business data.

character getProcedureName ()
For a callable statement this returns the name of the stored procedure requested to be executed, for other statements returns constant values used by the framework.

Callable Statement

ro.acorn.jdbc.sql.CallableStatement

When a business logic service is called from the JDBC client the stored procedure class that implements the service receive an instance of CallableStatement that holds all input parameters that were set by the client and which can also be used to set output/ return parameter values back to the client - the client need to use a callable statement for that and register the output parameters before making the call.

This object extends the base Statement object by adding functionality to retrieve and set parameters that can be passed between client and server and the other way around.

The data type of each parameter sent is important to match the data type expected for that parameter, data conversion can be done, if possible, when data types do not match. Parameters are retrieved using index position which is a one-base index (starting from 1). The callable statement provides methods to get or set parameters for various data types.

inherits Statement

integer getByte (parameterIndex as integer)
Retrieve the value of JDBC `TINYINT` or `BIT` parameter as integer data type.

memptr getBytes (parameterIndex as integer)
Retrieve the value of JDBC `BINARY` or `VARBINARY` parameter as a bytes array memory pointer data type.

logical getBoolean (parameterIndex as integer)
Retrieve the value of JDBC `BIT` or `BOOLEAN` parameter as logical data type.

longchar getClob (parameterIndex as integer)
Retrieve the value of JDBC `CLOB` parameter as longchar data type.

handle getDataset (parameterIndex as integer)
Retrieve a dataset handle by parsing the value of JDBC `VARCHAR` parameter that should contain the XML dataset serialization (a format that can be used with read-xml).

date getDate (parameterIndex as integer)
Retrieve the value of JDBC `DATE` parameter as date data type.

decimal getDouble (parameterIndex as integer)
Retrieve the value of JDBC `DOUBLE` or `REAL` or `NUMERIC` parameter as decimal data type.

decimal getFloat (parameterIndex as integer)
Retrieve the value of JDBC `FLOAT` or `NUMERIC` parameter as decimal data type.

integer getInt (parameterIndex as integer)
Retrieve the value of JDBC `INTEGER` parameter as integer data type.

int64 getLong (parameterIndex as integer)
Retrieve the value of JDBC `LONG` parameter as big integer data type.

integer getNumParams ()
Returns the number of parameters passed to this statement.

integer getShort (parameterIndex as integer)
Retrieve the value of JDBC `SMALLINT` or `TINYINT` parameter as integer data type.

integer getSQLType (parameterIndex as integer)
Return the SQL data type of the parameter.

character getString (parameterIndex as integer)
Retrieve the value of JDBC `CHAR` or `LONGCHAR` or `VARCHAR` parameter as character data type.

handle getTable (parameterIndex as integer)
Retrieve a temp-table handle by parsing the value of JDBC `VARCHAR` parameter that should contain the XML temp-table serialization (a format that can be used with read-xml).

datetime getTime (parameterIndex as integer)
Retrieve the value of JDBC `TIME` or `TIMESTAMP` parameter as date-

time data type.

datetime-tz getTimestamp (parameterIndex as integer)
Retrieve the value of JDBC `TIMESTAMP` parameter as date-time with time-zone data type.

void loadDataset (parameterIndex as integer, dataset-handle dsHandle)
Load data in a dataset handle by parsing the value of JDBC `VARCHAR` parameter that should contain the XML dataset serialization (a format that can be used with read-xml).

void loadTable (parameterIndex as integer, table-handle ttHandle)
Load data in a temp-table handle by parsing the value of JDBC `VARCHAR` parameter that should contain the XML temp-table serialization (a format that can be used with read-xml).

void setByte (parameterIndex as integer, parameterVal as integer)
Set the value of JDBC `BIT` parameter.

void setBytes (parameterIndex as integer, parameterVal as memptr)
Set the value of JDBC `BINARY` parameter.

void setBoolean (parameterIndex as integer, parameterVal as logical)
Set the value of JDBC `BOOLEAN` parameter.

void setClob (parameterIndex as integer, parameterVal as longchar)
Set the value of JDBC `CLOB` parameter.

void setDate (parameterIndex as integer, parameterVal as date)
Set the value of JDBC `DATE` parameter.

void setDouble (parameterIndex as integer, parameter as decimal)
Set the value of JDBC `DOUBLE` parameter.

void setFloat (parameterIndex as integer, parameterVal as decimal)
Set the value of JDBC `FLOAT` parameter.

void setInt (parameterIndex as integer, parameterVal as integer)
Set the value of JDBC `INTEGER` parameter.

void setLong (parameterIndex as integer, parameterVal as

int64)
Set the value of JDBC `LONG` parameter.

void setShort (parameterIndex as integer, parameterVal as integer)
Set the value of JDBC `TINYINT` parameter.

void setString (parameterIndex as integer, parameterVal as character)
Set the value of JDBC `VARCHAR` parameter.

void setTime (parameterIndex as integer, parameterVal as datetime)
Set the value of JDBC `TIME` parameter.

void setTimestamp (parameterIndex as integer, parameterVal as datetime-tz)
Set the value of JDBC `TIMESTAMP` parameter.

Select Statement

ro.acorn.jdbc.sql.SelectStatement

When a data query is sent from the JDBC client the all query details are loaded in an select statement instance - query buffers and fields, where clause(s) and pagination options.

This object extends the base Statement object and is usually used by the framework to select data directly from database table(s) but it can also be used for business views when a custom 'business view' of the data is built on top of existing tables - like doing de-normalisation and field aggregation.

inherits Statement

integer getFetchSize ()
Returns the preferred client page fetch size. The standard select statement can use pagination, for business views this is only important if the view also support pagination by implementing the IBufferedView interface.

character getFieldAlias (bufferIndex as integer, fieldIndex as integer)
Returns the alias for the fields at given position in one of the query buffers, if no alias is used this will return the field name. Both buffer and field indexes are 1-base.

character getFieldName (bufferIndex as integer, fieldIndex as integer)
Returns the name for the fields at given position in one of the query buffers. Both buffer and field indexes are 1-base.

`integer getLimit ()`
Returns the maximum number of records the client expects to receive back, used for data pagination.

`integer getNumBuffers ()`
Returns the number of buffers used in select query, for business views only a single table/view select is supported so this will always be 1 in that case.

`integer getNumFields (bufferIndex as integer)`
Returns the number of fields selected from that particular query buffer.

`integer getOffset ()`
Returns the record offset from which the records retrieval should start, used for data pagination.

`character getOrderBy ()`
Returns the order by clause used for current query if any.

`character getOrderBy (fieldMap as character)`
Returns the order by clause used for current query if any using a field map list for name resolution. This can be used when the field names from the business view doesn't match the one in underlying table(s) to obtain a valid order-by clause that can be used in queries against the database table(s). The field map is a comma separated list of field and alias pairs.

`character getWhereClause ()`
Returns the where clause used for current query if any.

`character getWhereClause (fieldMap as character)`
Returns the where clause used for current query if any using a field map list for name resolution. This can be used when the field names from the business view doesn't match the one in underlying table(s) to obtain a valid where clause that can be used in queries against the database table(s). The field map is a comma separated list of field and alias pairs.

`logical hasAggregateFields ()`
Returns true if any aggregate function is used for a field in current query.

`logical hasNonAggregateFields ()`
Returns true if there are fields selected for which no any aggregate function is used in current query.

`logical isAllFields (bufferIndex as integer)`
Returns true if all fields from that particular query buffer are selected - star (*) was used to return all fields.

`logical isBreakBy ()`
Returns true if break option was used with the order by clause.

logical isDistinctOnly ()
Returns true if only distinct records should be retrieved based on the order by clause.

logical isExceptFields (bufferIndex as integer)
Returns true if fields listed for a particular query buffer are to be excepted - this means all other fields from the buffer are to be selected instead.

SQLState

ro.acorn.jdbc.sql.SQLState

Enumeration class holding all driver specific SQL states.

private constructor SQLState ()

The class has a private constructor, enumeration values are only to be accessed in a statical manner.

SUCCESSFUL_COMPLETION	
GENERAL_ERROR	
GENERAL_WARNING	
NO_DATA	
SYNTAX_ERROR	
NOT_AUTHORIZED	
NO_PRIVILEGE	
TABLE_NOT_FOUND	
COLUMN_NOT_FOUND	
MISSING_PARAMETERS	
WRONG_TYPE_PARAMETERS	
DATA_EXCEPTION	
DATA_NUMERIC_OVERFLOW	
DATA_DATETIME_OVERFLOW	
DATA_DATETIME_INVALID	
DATA_DIVISION_BY_ZERO	
DATA_ASSIGN_ERROR	
DATA_DUPLICATE_ENTRY	
INVALID_CHARACTER_SET	
INVALID_CATALOG	
INVALID_SCHEMA	
INVALID_DESCRIPTOR	
FEATURE_NOT_SUPPORTED	
WRONG_VALUE_COUNT	

SQLException

ro.acorn.jdbc.sql.SQLException

When errors occurs when executing a statement those can be sent back to the client by simply adding them to the statement's exception or warning list. Since batch and select statements are usualy handled by the framework this is often used in the callable statement case when stored procedures are executed.

inherits Progress.Lang.AppError

constructor SQLException (reason as character)
Create a new exception instance using the SQLState "generic error", the main exception 'reason' is set by the first string parameter.

constructor SQLException (errorObj as Progress.Lang.Error)
Create a new exception instance by retrieving all messages from the error object using the SQLState "generic error", first error message is used as 'reason'.

constructor SQLException (reason as character, errorObj as Progress.Lang.Error)
Create a new exception instance by retrieving all messages from the error object using the SQLState "generic error", the main exception 'reason' is set by the first string parameter.

constructor SQLException (reason as character, nativeCode as integer)
Create a new exception instance using the SQLState "generic error", the main exception 'reason' is set by the first string parameter and second parameter is the native error code - the Progress error code.

constructor SQLException (reason as character, nativeCode as integer,
sqlState as character)
Create a new exception instance, the main exception 'reason' is set by the first string parameter, second parameter is the native error code and last is the SQL state code. The SQL state can actually be anything but there is a list of most common error types that can be used.

integer getErrorCode ()
Returns exception native error code, can be either Progress own error numbers or application specific one.

static character getErrorMessage (proError as Progress.Lang.Error)

Returns the main error message from the Progress error object - this is a convenience method to solve the inconsistency between AppError and other Progress error objects.

`character getMessage ()`
Returns the main error message of this exception, this is not the root cause but the reason used when the exception was instantiated.

`SQLException getNextException ()`
Returns the cause of this exception if any, exceptions can be chained that way to get more details about what went wrong when executing a statement.

`character getSQLState ()`
Returns the SQL state code of the exception.

`void setNextException (sqlErr as SQLException)`
Add a new exception on the exception chain as causes of this given exception.

SQLWarning

`ro.acorn.jdbc.sql.SQLWarning`

This is just a special case of an SQLException that is meant to only signal a warning condition not an actual exception, it inherits all constructors from the base class.

`inherits SQLException`

Services

A number of core services are available and depending on configuration options can be activated or not, the services are build to be plug-able and can be easily extended if needed.

Services that are currently available and can be activated by simply changing the configuration are: authentication, authorization, business catalog, logging and configuration service itself.

Please note that the core authentication services provided both use the client principal object and as such the new audit features of Open Edge are going to trap each user action if activated, however the logging service can also be activated for audit purposes if built-in Open Edge audit feature is not activated.

- Authentication
- Authorization
- Business Catalog
- Configuration

- Logging

Some services can be configurable which means it implements the IConfigurable interface and when the controller loads them it will automatically call the load configuration method on them passing the configuration service manager and the appropriate configuration section. If a class hierarchy is create for a particular service then the specific service configuration can override the generic service configuration, meaning the service should look first for specific configuration in given service configuration section and if not use the generic one.

Service configuration section for those that support configuration is in service specific section under: `/sql/services`.

Authentication

The authentication service is used in cases where access to the business logic is restricted to authenticated users.

There are a number of already implemented service managers for that provided by the framework but there is also an option to use a custom service manager if needed proved that it implement the required interface - IAuthentication.

For services that relies on using the client principal object for passing the authentication token an abstract base class is provided with common functionality - AuthenticationBase.

For service managers implementing the IConfigurable interface the configuration section used by default is: `/sql/services/authentication`.

Note: In order to support user management related Data Definition Language (DDL) the interface will at some point be extended to support functions like: CREATE, DROP, ALTER user.

IAuthentication

`ro.acorn.jdbc.service.IAuthentication`

This is the interface that any authentication service manager must implement in order to be used as such by the framework.

`raw getToken (userInfo as ClientPrincipal)`

This method seals the client principal object containing user information and returns the authentication token that is going to be used by all subsequent requests made within the same user session. The authentication token is usually the encrypted serialization of the client principal but it might also be a simple session identifier.

ClientPrincipal login (userName as character, userPasswd as character)
This method validates the user credentials passed and if valid it returns the authentication token that is going to be used by all subsequent requests made within the same user session. The authentication token can be anything from the full session object serialization to a simple session identifier.

ClientPrincipal parseToken (authenticationToken as raw)
This method validates the authentication token, if the authentication token is not valid the user request will be rejected. The authentication service can also implement a session time-out mechanism and invalidate a token after a certain time or inactivity.

As this method is being called on each user request this is the place where the authenticated user can be set either for the whole session (`SET-CLIENT`) or for specific database(s) (`SET-DB-CLIENT`) for audit purposes.

If the application business logic needs some sort of session state this might also be a good place to restore user's session.

ClientPrincipal

ro.acorn.jdbc.service.authentication.ClientPrincipal

This is merely a wrapper on top of a client-principal object handle and is used for authentication and authorization of SQL clients. On each connection the user credentials are validated and a user session is established, details about the user and session information is stored in a client-principal object that is then serialized back to the client. On every request sent from the client the session information is passed back and restored in a ClientPrincipal instance that is accessible using `getClientInfo` method either from the CallableStatement injected as parameter when a stored procedure runs or statically from the Controller.

```
static ClientPrincipal createClient ( userName as
character,
     authDomain as character, sessionId as character)
```

Create a ClientPrincipal instance using the client and session information provided, if first parameter contains domain information (user@domain) that takes precedence over the one set as second parameter. The instance is not 'sealed' so additional information can be added using `setProperty` method.

```
static raw createToken (userName as character,
     authDomain as character, sessionId as character,
     sealKey as character)
```
Create a ClientPrincipal instance using the client and session information provided, 'seal' the instance using the provided secret

key and return the session token that can be used for session management afterwards. This can be used when no additional information need to be set using `setProperty` method.

`static ClientPrincipal loadClient ( sessionToken as raw,`
`sealKey as character)`
Create a ClientPrincipal instance by decrypting the session token, the token 'seal' is validated using the provided secret key and error is thrown if secret key do not match or the token was been tampered with. This is normally done on each client request to re-establish the user session.

`character getDomain ()`
Returns the domain name set on the client principal instance.

`datetime-tz getSessionExpire ()`
Returns the session expiration timestamp if set, if session was not set to expire this will return null.

`character getProperty (propertyName as character)`
Returns the value of a property set for the session, returns null if property value not set.

`character getQualifiedName ()`
Returns the full used name using the 'user@domain' format, if no domain is set this returns just the user name.

`datetime-tz getSessionId ()`
Returns the session unique identifier, this can be used with an external session manager service to keep additional session state (user code, language code, currency code, time zone, etc).

`datetime-tz getSessionStart ()`
Returns the timestamp of when the session started, the login process is considered to be finished when the client principal was sealed.

`raw getToken ()`
Returns a portable security token regardless if information was already sealed or not.

`raw getToken (secretKey as character)`
Returns a portable security token after sealing the client information using the given secret key as domain access code.

`character getUser ()`
Returns the used name.

`void seal (secretKey as character)`
Seals the client information using the given secret key as domain access code, if already sealed this throws an error.

`void seal (secretKey as character, expireTime as`

datetime-tz)
Seals the client information using the given secret key as domain access code and set a session expiration time, if already sealed this throws an error.

void setDbClient (databaseName as character)
Set the user identity for given database (logical name) using user information set in this instance.

void setProperty (propertyName as character, propertyValue as character)
Set the value of a property into the session context. Because this whole client context is serialised and passed back and forth on every request is better to keep the number of session properties stored in it to the minimum - use the session identifier together with a proper session management service that stores session data in a database table as an alternative.

Authentication Base

ro.acorn.jdbc.service.authentication.AuthenticationBase

This is an abstract class that implements some basic functionality based on client principal object which once sealed is being passed along as an authentication token, not all methods from IAuthentication interface are implemented. The class can be used as a base class for any authentication service that want to use the same client principal approach in which case only the `login` method need to be implemented.

The class has also a number of extra methods that basically set a number of options like authentication domain, and password encryption options; the class is also configurable (implements IConfigurable interface) meaning it can use a configuration service for auto-configuration.

character getDomain ()
Returns the authentication domain used.

character getSecretKey ()
Returns the authentication secret key used to seal and validate the user information into the client principal.

raw getToken (userInfo as ClientPrincipal)
This method seals the client principal object containing user information and returns the authentication token that is going to be used by all subsequent requests made within the same user session. The authentication token is usually the encrypted serialization of the client principal but it might also be a simple session identifier.

ClientPrincipal parseToken (authenticationToken as raw)
This method validates the authentication token, if the authentication

token is not valid the user request will be rejected.

void setDomain (domain as character)
Set the authentication domain used to seal the client principal object. The domain need to be already registered in the session security policy using the specified domain secret key.

void setSecretKey (domainKey as character)
Set the authentication secret key used to seal the client principal object.

All authentication options can be configured in main authentication service configuration section using the following keys:

`domainName`	Authentication domain.
`secretKey`	Authentication secret key.

Anonymous Authentication

ro.acorn.jdbc.service.authentication.AnonymousAuthentication

This is the authentication service used when no authentication is actually enforced, just to be able to keep session state.

inherits AuthenticationBase

ClientPrincipal login (userName as character, userPassword as character)

Returns a client principal instance using provided user name - or 'anonymous' if not specified.

Database Authentication

ro.acorn.jdbc.service.authentication.DatabaseAuthentication

This is basic database authentication service which is using the default database authentication mechanism or Progress database.

inherits AuthenticationBase

character getDatabase ()

Returns the database used for authentication.

ClientPrincipal login (userName as character, userPassword as character)

Validate the user and password against database's users and return a client principal instance if valid credentials or throw an error otherwise.

void setDatabase (database as character)

Set the database used to authenticate users against.

The database to be used for authentication can be configured in service specific configuration section "*sql/services/authentication*":

```
    databaseAuthentication/databaseName
```

Domain specific authentication options can override the ones from main authentication service configuration section by specifying them in the service specific configuration section:

```
    databaseAuthentication/domainName
    databaseAuthentication/domainKey
```

The encryption options are not applicable as the database security mechanism is going to be used directly.

Table Authentication

ro.acorn.jdbc.service.authentication.TableAuthentication

This is basic table authentication service which is using a application specific user table to validate login credentials against.

inherits AuthenticationBase

The kind of possible encryption methods are defined as static constants:

`encryptNone`	password is stored unencrypted
`encryptEncode`	password is encrypted using one-way encode method
`encryptMD5`	password is encrypted with the MD5 hash algorithm using the provided key as hash key
`encryptSHA1`	password is encrypted with the SHA1 hash algorithm using the provided key as hash key

character getEncryptKey ()
Returns the key used to encrypt the user password when storing it

into the user table, not used when encryption method selected is encode.

`integer getEncryptMethod ()`
Returns the method used to encrypt the user password when storing it into the user table.

`character getPasswordField ()`
Returns the name of table field that holds the user password.

`character getTable ()`
Returns the name of the user table used for authentication.

`character getUserField ()`
Returns the name of table field that holds the user name/id.

`ClientPrincipal login (userName as character, userPassword as character)`
Validate the user and password against users table and return a client principal instance if valid credentials or throw an error otherwise.

`void setEncryptKey (key as character)`
Set the hash key to be used when user password is encrypted to be stored into the user table.

`void setEncryptMethod (encryptMethod as integer)`
Set the algorithm to be used when user password is encrypted to be stored into the user table.

```
void setTable (table as character,
     userField as character,
     passwdField as character)
```
Set the information about the user table used to authenticate users against. Table name should be qualified with the database name if needed, user and password field should be valid fields on the given user table.

The information about the user table to be used for authentication and encryption options can be configured in service specific configuration section "*sql/services/authentication*":

```
tableAuthentication/tableName
tableAuthentication/userField
tableAuthentication/passwordField
tableAuthentication/encryptMethod
tableAuthentication/encryptKey
```

The default encryption method used if not otherwise specified is `encryptEncode`.

Authorization

The authorization service is used in cases where access to the business logic services require authorization.

Currently there is no authorization service provided by the framework but there is always possible to use a custom service manager if needed proved that it implement the required interface - IAuthorization.

For service managers implementing the IConfigurable interface the configuration section used by default is: `/sql/services/authorization`.

Note: In order to support Data Control Language (DCL) the interface will at some point be extended to support functions like: GRANT/REVOKE and require authorization groups/roles.

IAuthorization

`ro.acorn.jdbc.service.IAuthorization`

This is the interface that any authorization service manager must implement in order to be used as such by the framework.

```
logical isAllowed (userInfo as ClientPrincipal,
functionName as character)
```

This method validates the user privileges for performing specified function, it returns true if the user has granted the privilege and false otherwise.

The authorization service manager might implement group/roles based authentication or not, the authorization service manager might need to implement additional interfaces if it supports Data Control Language statements and group/roles authorization.

Business Catalog

The Business Logic Catalog Service is managing the business catalog and provides meta-data information about registered services - business logic procedures, and business views.

How the catalog service is implementing procedures and views registration or where and how it stores meta-data information is not relevant for the framework, therefore the ICatalog interface that need to be implemented by a custom catalog service only contains methods for querying the catalog information.

There is an abstract base class that can be used to more easily implement a custom service and the framework provides a default implementation using a file data-store.

For service managers implementing the IConfigurable interface the configuration section used by default is: `/sql/services/businessCatalog`.

ICatalog

`ro.acorn.jdbc.service.ICatalog`

This interface defines the public methods a business catalog registration service need to implement.

`IStoredProcedure getProcedure (procName as character)`
Returns an instance of the class that implements the requested business service, it's up to the registration service what the behaviour is when the service is registered on multiple catalogs.

`IStoredProcedure getProcedure (procName as character, catalog as character)`
Returns an instance of the class that implements the requested business service and was registered for the catalog. If no class providing that business service was registered on given catalog null is returned.

```
void getProcedureColumns (catalog as character,
procNamePattern as character,
        columnNamePattern as character, output table-
    handle procedureColumnsSet)
```
Returns the list of procedure's parameters, this includes columns from procedure's return set(s); if catalog parameter is null the search will be done on all catalogs else equality match is to be used, both procedure and column name parameters are regular expression patterns that can be used together with *matches* operator. The output temp-table should have the following structure:

Field name	**Data type**	**Description**
p_catalog	character	Catalog name where the procedure is registered
p_name	character	Procedure name, not the class name implementing it but the 'business' name of this procedure/service.
c_position	integer	Column index in procedure's result set(s).
c_name	character	Column name
c_type	integer	Column type (input, output, input-output, result set) - valid values in ParameterMetaData static properties.

c_data_type	character	Column data type - Progress data types not SQL equivalent.
c_desc	character	Column description
c_null	integer	Column nullable settings - valid values in ParameterMetaData static properties.

handle getProcedureResultMeta (procName as character, catalog as character,
resultIndex as integer)
Returns the table-handle of specific procedure's result set, if any. If procedure returns a dataset then resultIndex parameter is used to return a particular table from it - 1 base index, if procedure does not return any result set(s) or resultIndex is invalid the method returns *null*.

void getProcedures (catalog as character, procNamePattern as character,
output table-handle procedureCatalogSet)
Returns the list of procedures registered with the catalog service, if catalog parameter is null the search will be done on all catalogs else equality match is to be used, procedure name parameter can be a regular expression pattern that can be used together with *matches* operator. The output temp-table should have the following structure:

Field name	**Data type**	**Description**
p_catalog	character	Catalog name where the procedure is registered
p_name	character	Procedure name, not the class name implementing it but the 'business' name of this procedure/service.
p_desc	character	Procedure description
p_type	integer	Procedure return type - valid values in ProcedureMetaData static properties.

IView getView (viewName as character)
Returns an instance of the class that implements the requested business view, it's up to the registration service what the behaviour is when the view is registered on multiple catalogs.

IView getView (viewName as character, catalog as character)
Returns an instance of the class that implements the requested business view and was registered for the catalog. If no class providing that business view was registered on given catalog null is returned.

void getViewColumns (catalog as character, viewNamePattern as character, columnNamePattern as character, output table-handle viewColumnsSet)

Returns the list of view's columns; if catalog parameter is null the search will be done on all catalogs else equality match is to be used, both view and column name parameters are regular expression patterns that can be used together with *matches* operator. The output temp-table should have the following structure:

Field name	**Data type**	**Description**
v_catalog	character	Catalog name where the view is registered
v_name	character	View name, not the class name implementing it but the 'business' name of this 'view'
c_position	integer	Column index
c_name	character	Column name
c_data_type	character	Column data type - Progress own data types
c_desc	character	View's column description
c_null	logical	True if column can be null, false if mandatory
c_decimals	integer	Number of decimals, for decimal fields
c_width	integer	Column's SQL width
c_default	character	Column's default value (buffer field's string-value)

void getViewIndexes (catalog as character, viewNamePattern as character, uniqueFlas as logical, primaryFlag as logical, output table-handle viewColumnsSet)

Returns index information about registered view(s), includes fields of each index; if catalog parameter is null the search will be done on all catalogs else equality match is to be used, both view name parameter is a regular expression pattern that can be used together with *matches* operator. If unique and primary flags are not null only unique, respectively primary indexes are to be returned. The output temp-table should have the following structure:

Field name	**Data type**	**Description**
v_catalog	character	Catalog name where the view is registered
v_name	characte	View name, not the class name implementing

	r	it but the 'business' name of this 'view'
i_position	integer	Index position
i_name	character	Index name
i_unique	logical	True if index is unique
i_primary	logical	True if index is primary
f_position	integer	Index field position inside the index
f_name	character	Index field name
f_desc	logical	True if field values are indexed in a descending order

void getViews (catalog as character, viewNamePattern as character,
    output table-handle viewCatalogSet)

Returns the list of views registered with the catalog service, if catalog parameter is null the search will be done on all catalogs else equality match is to be used, view name parameter can be a regular expression pattern that can be used together with *matches* operator. The output temp-table should have the following structure:

Field name	**Data type**	**Description**
v_id	character	Unique view identifier
v_catalog	character	Catalog name where the view is registered
v_name	character	View name, not the class name implementing it but the 'business' name of this 'view'.
v_desc	character	View description

Catalog Base

`ro.acorn.jdbc.service.catalog.CatalogBase`

This is an abstract base class that can be used to extend the framework by providing a custom business catalog service manager. This base class implements all methods required by the ICatalog service interface and a number of methods for registering/de-registering views and procedures and it's configurable using the settings explained in the configuration section.

Catalog service manager implementations extending this base class need, at minimum, to implement two abstract methods: `loadCatalog` and `saveCatalog`. Business catalog information can be persisted in any form (flat file, JSON, database tables) and it's the implementation's responsibility to persist that information down to the storage level while doing any serialization/de-serialisation needed.

Configuration

Any service extending this base class can be configured through the configuration service by setting following keys in "*sql/services/ businessCatalog*" section, the list of available configuration options can be found on the configuration page.

`logical isRegistrationAllowed (path as character)`
Returns true if classes - procedures and views - from given path can be registered into the business catalog. Only paths, and sub-paths, from configured CLASSPATH can be registered.

`void loadCatalog ()`
Abstract method for loading business catalog information from persistence layer into internal temp-tables (protected so can be accessed from classes extending this base class).

`void registerAll ()`
Registers all classes implementing IStoredProcedure or IView into default catalog - the currently selected database - by recursively register every folder from CLASSPATH.

`void registerAll (catalog as character)`
Registers all classes implementing IStoredProcedure or IView into given catalog by recursively register every folder from CLASSPATH.

`void registerFolder (path as character)`
Registers all classes implementing IStoredProcedure or IView from given folder into default catalog - the currently selected database. The folder should be under CLASSPATH in order to be able to be registered.

`void registerAll (path as character, catalog as character)`
Registers all classes implementing IStoredProcedure or IView from given folder into specific catalog. The folder should be under CLASSPATH in order to be able to be registered.

`void registerProcedure (className as character, catalog as character)`
Registers the class into given catalog proving it's implementing a business service - the class must implement the StoredProcedure interface. The same business service can be registered on more than one catalog, though the business service name need to be unique for the catalog. The business service name is not the name of the class that implement it but the name under which the service register itself - the name returned by the business service meta data instance.

`void registerProcedure (className as character)`

Registers the class implementing IStoredProcedure into default catalog - the currently selected database, class name should be fully qualified and available in CLASSPATH.

void registerProcedure (businessService as IStoredProcedure, catalog as character)
Registers the business service into given catalog, if autoSave is set to true the catalog will be automatically saved if registration succeeds.

void registerProcedure (businessService as IStoredProcedure)
Registers the business service into default catalog - the currently selected database.

void registerView (className as character)
Registers the class implementing IView into default catalog - the currently selected database, class name should be fully qualified and available in CLASSPATH.

void registerView (businessView as IView, catalog as character)
Registers the business view into given catalog, if autoSave is set to true the catalog will be automatically saved if registration succeeds.

void registerView (businessView as IView)
Registers the business view into default catalog - the currently selected database.

void saveCatalog ()
Abstract method to save the business catalog from internal temp-tables to the persistence layer.

void unregisterProcedure (className as character, catalog as character)
Remove a registered class from given catalog, the input parameter is actually the class type name that is implementing a business service not the business service name.

void unregisterProcedure (className as character)
Remove a registered class from default catalog - the currently selected database.

void unregisterProcedure (businessService as IStoredProcedure, catalog as character)
Remove the registered business service from given catalog.

void unregisterProcedure (businessService as IStoredProcedure)
Remove the business service from default catalog - the currently selected database.

`void unregisterView (className as character, catalog as character)`
Remove a registered class from given catalog, the input parameter is actually the class type name that is implementing the business view not the business view name.

`void unregisterView (className as character)`
Remove a registered class from default catalog - the currently selected database.

`void unregisterView (businessView as` IView**`, catalog as character)`**
Remove the registered business view from given catalog.

`void unregisterView (businessView as` IView**`)`**
Remove the business view from default catalog - the currently selected database.

Configuration

Business catalog configuration is done through the common configuration service in a dedicated configuration section *"/sql/services/businessCatalog".*

The following configuration options are available:

- classPath

  Default is empty, this should be updated with the list of folders that contains business services that can be registered. The list is comma separated and each folder path need to be relative to the Application Server PROPATH. If the business catalog registration service is set to work in auto refresh mode the number of entries in the class path can affect the performance of registration process.

  For example if the application PROPATH include two folders: `/usr/app/crm` and `/usr/app/erp` and the business logic services to be registered are located in `/usr/app/crm/sql` and `/usr/app/erp/sql` then a single entry need to be added in class path which is '`sql`'.

- autoSave

  Default is false, this direct the business catalog registration service to update the business catalog persistence storage each time the business catalog is updated - new procedure/view registered, updated or deleted (de-register).

  If this is set to false then all changes made to the business catalog that were not saved manually will be lost when the Application Server is restarted.

- autoRefresh

  Default is false, this can be used in development environment where more business procedures/views are added or gets updated very often and cause the business catalog registration service to auto register when requested. When a request is made for a business procedure which is not yet registered in the catalog then the registration service will search for the class that implements the requested business procedure through all class path and if found it register it for further use.

  If is set to false then when a request is made for a business procedure which is not registered in the catalog an error condition is raised, although the class that implements the business procedure might exist the registration service does not look for it.

File Catalog

ro.acorn.jdbc.service.catalog.FileCatalog

The business catalog is stored in a plain XML file that is loaded at start-up and can be updated manually or automatically when the business catalog is updated, new services are registered or old services removed or updated.

In order for a business procedure or view to be registered the class need to be located somewhere in the class path configured for the business catalog, there is also an auto refresh feature that if set will allow auto registration of services on request proven the class that implements the requested service is found in the business catalog class path.

Configuration

Catalog service configuration options can be set in *businessCatalog* service configuration section - "*sql/services/businessCatalog*"

Key name	Description	Default v
`catalogFile`	XML file where catalog is stored.	"*ro/acorn*
`encryptionMode`	If the information is to be stored encrypted set this to any valid encryption algorithm (symmetric-encryption-algorithm).	"none"
`encryptionMode`	If encryption is used this is the password to be used to encrypt catalog information when saved.	

inherits CatalogBase

The following public methods of the business catalog registration service can be used.

`void loadCatalog (filePath as character)`
Loads the business catalog from given catalog file.

`void loadCatalog ()`
Loads the business catalog from current XML catalog file, default XML catalog file is "`ro/acorn/jdbc/config/catalog.xml`".

`void saveCatalog (filePath as character)`
Saves the business catalog to given XML catalog file.

`void saveCatalog ()`

Saves the business catalog into current XML catalog file.

Configuration

The configuration service used by the framework is using XPATH over a plain XML configuration file. Although there is a configuration service interface definition `IConfiguration` for now the framework only provides one implementation - the XML based configuration service. If the 4GL application already have a configuration service - that might store information in a database for instance - a wrapper to that can be created by implementing the required interface. Still, at start-up the XML configuration file will be used and if an alternate configuration service is configured then the framework's default service will be swapped with the alternate one.

The configuration service can be retrieved from the controller object and it's method can be called directly, however for framework services is better to implement the configurable interface `IConfigurable` which will make the controller call the load configuration method with the appropriate configuration service when loading the service manager.

IConfiguration

ro.acorn.jdbc.service.IConfiguration

This is the interface that any configuration service manager must implement in order to be used as such by the framework.

`void close ()`
Close the currently loaded configuration content, no configuration context is available until another bundle is loaded.

`character getProperty (key as character)`
Retrieve the value of given configuration key, the key should be of a form of an XML fully-qualified path starting from the configuration root element:

```
/root/section/[/section...]/key
```

If the key is not found then null is returned, depending on the configuration service implementation the key name might be case sensitive or not.

`character getProperty (key as character, default as character)`
Retrieve the value of given configuration key, if key isn't found the default value is returned.

`void load (url as character)`
Load the configuration content from given URL, this can be a plain

file-system path for file based configuration services or any other URL format used by the specific service manager to locate the resource bundle to load.

void save (url as character)
Save the current configuration content into given URL, this can be a plain file-system path for file based configuration services or any other URL format used by the specific service manager to locate the resource bundle to save into.

character setProperty (key as character, value as character)
Set the value of given configuration key, if key already existed the previous key value is returned, null otherwise.

IConfigurable

ro.acorn.jdbc.service.IConfigurable

This is the interface that need to be implemented by any object that can gets it's configuration from a configuration service manager. It is used basically by the framework service managers that retrieve their configuration options from the framework's core configuration service.

```
void configure (configService as IConfiguration,
                configSection as character)
```

The configurable object sets default values for any configurable properties by loading the corresponding key values from given configuration service manager and the configuration section specified. An object might choose to look for configuration values only in the provided section, look on specific section or all sections under it or even search for them through the whole configuration tree.

XML Configuration

ro.acorn.jdbc.service.configuration.XMLConfiguration

The default configuration service manager provided uses a plain XML file as persistence storage.

void load (xmlFile as character)
Load the configuration content from given XML file.

void save (xmlFile as character)
Save the current configuration content into given XML file.

void setCaseInsensitive (caseInsensitive as logical)
Set the case sensitive option to be used when looking-up configuration key names, if set to true a key is considered to be the same even if it's name casing differs - the initial value is true, case

insensitive keys.

Logging

The logging service is used by the framework services or from the stored procedures to log additional information.

Currently there is a default logging service provided by the framework that uses the log-manager object for logging messages into the application server's agent log file, a custom loggin service can be creating by implementing the ILogger interface.

For service managers implementing the IConfigurable interface the configuration section used by default is: `/sql/services/logging`.

ILogger

`ro.acorn.jdbc.service.logging.ILogger`

This is the interface that any logging service manager must implement in order to be used as such by the framework.

`void logDebug (message as character)`
Log the respective debug message, proven the log level is set to include those messages.

`void logError (message as character, error as Progress.Lang.Error)`
Log the respective error message plus details about the exception that caused the error if any provided, error messages should be recorded regardless of the log level.

`void logInfo (message as character)`
Log the respective informative message, proven the log level is set to include those messages.

`void logWarn (message as character, error as Progress.Lang.Error)`
Log the respective warning message plus details about the exception that caused the error if any provided, proven the log level is set to include those messages.

LogManager Logger

`ro.acorn.jdbc.service.logging.LogManagerLogger`

The default logging service manager provided uses the log-manager object to log messages into application server's log files, implements all methods of ILogger interface and is configurable.

`integer getLogLevel ()`
Returns the current log level, initial log level is errors only unless changed through configuration using property `/sql/services/logging/logLevel` - valid values: ERROR, WARN, INFO, DEBUG.

`void setLogLevel (logLevel as integer)`
Set the log level, must be greater than zero else current log level remains unchanged.

Business View

`ro.acorn.jdbc.sql.IView`

In order to be able to expose the business logic as a service this simple interface need to be implemented by each class that is going to be registered in the business logic catalog. If your business logic supports pagination then is better to also implement the IBufferedProcedure interface.

`ProcedureMetaData getMetaData`

The business logic service need to be able to self-describe itself in order for the business catalog registration service to register it.

The method should return a valid instance of ProcedureMetaData object that can be obtained by simply calling the constructor providing the procedure name which is mandatory and optionally the description of the business logic service as it's going to appear on business catalog. If the stored procedure does not return a valid instance of procedure meta data or if the service name meta data information was not set then the procedure fails to be registered into the business catalog.

If the business logic service accepts parameters (input, input-output or output), returns a value or a result set there are methods in ProcedureMetaData object to describe those parameters as well. Although this is not mandatory it can provide more details on how the business logic service (stored procedure) need to be called and what the result looks like.

```
integer executeProcedure
    (callStatement as CallableStatement,
     output resultSet as handle)
```

This method is called when a store procedure call statement is made from the JDBC client; normally this should call some service of existing application business logic and depending on input parameters received it can set output parameters and return a primitive data type, a temp-table or dataset handle.

The method have two parameters, first is an instance of CallableStatement that can be used to retrieve the input parameters sent from JDBC client and set values of output parameters; last parameter of the method is an output handle to a temp-table or a dataset containing the result set(s).

Since the output parameter can be either a temp-table or dataset a simple handle is used instead of table-handle or dataset-handle, this means that the data structure must still be valid (not be deleted if dynamically created) when the method ends. It is expected however that, if dynamic, the result data structure to be deleted by the stored procedure's destructor in order to avoid memory leaks.

Standard JDBC does allows for multiple result sets and the ABL JDBC driver supports multiple open result sets.

View Meta Data

ro.acorn.jdbc.sql.metadata.ViewMetaData

This is a way for a business view that is going to be registered in the business logic catalog to describe itself by providing some useful meta-data information like description and data structure information - columns and indexes.

```
ViewMetaData (viewName as character,
     viewDescription as character,
     table-handle viewResultset)
```

Public constructor which creates a meta data object for the given business view name having an optional view description while the table handle defines the result set data structure. The view name need to be unique for the catalog in order to be able to register it in the business catalog.

ViewColumnMetaData getIndex (columnNum as integer)
Returns the meta-data information about the business view column.

character getDescription ()
Returns the business view description as is going to be shown in business catalog.

ViewIndexMetaData getIndex (indexNum as integer)
Returns the meta-data information about the business view index.

character getName ()
Returns the business view name as is going to be shown in business catalog.

integer getNumColumns ()

Returns the number of columns from the business view result set.

integer getNumIndexes ()
Returns the number of indexes defined on the business view result set.

View Column Meta Data

ro.acorn.jdbc.sql.metadata.ViewColumnMetaData

Used to describe each business view column, this is only used by the framework when a business view is registered into the business catalog.

ViewColumnMetaData
(name as character, dataType as character,
description as character, nullable as logical,
default as character, decimals as integer, width as integer)

Public constructor which creates a meta data object for a view column.

- name is mandatory
- data type is the Progress data type (translated to SQL equivalent by the DatabaseMetaData)
- description is optional but is preferable to be filled in
- nullable, if true the field can have a null value
- default, field initial default value
- decimals, number of decimal places for decimal fields only
- sql width, optional (inferred based on data type if not set)

character getDataType ()
Returns the business view field data type (4GL native data type).

integer getDecimals ()
Returns the number of decimal places used for the business view field, for non decimal fields returns 0.

character getDefault ()
Returns the default initial value of the business view field.

character getDescription ()
Returns the business view field description as is going to be shown in business catalog.

character getName ()
Returns the business view index name as is going to be shown in business catalog.

integer getWidth ()

Returns the SQL width of the business view field.

logical isNullable ()
Returns true if business view field value can be null.

View Index Meta Data

ro.acorn.jdbc.sql.metadata.ViewIndexMetaData

Used to describe each business view index, this is only used by the framework when a business view is registered into the business catalog.

ViewIndexMetaData (indexInformation as character)

Public constructor which creates a meta data object for a view index, the only input parameter required is a comma separated index information - as returned by index-information method on a table buffer.

character getFieldName (columnNum as integer)
Returns the name of field at given position in index's fields list.

character getName ()
Returns the business view index name as is going to be shown in business catalog.

integer getNumFields ()
Returns the number of fields on the business view index fields list.

logical isFieldDescending (columnNum as integer)
Returns true if field at given position in index's fields list is sorted in a descending order.

logical isPrimary ()
Returns true if business view's index is primary.

logical isUnique ()
Returns true if business view's index is unique.

Buffered View

ro.acorn.jdbc.sql.IBufferedView

Business views that supports pagination need to implement the IBufferedView interface, in which case the `selectData` method is called each time the client request another data page until `hasMoreData` returns false - the view instance need to keep track of last page retrieved as the SelectStatement passed is always the same as the one initially passed when the method was first called.

As with the business procedures, the fetch size set on the select statement can be interpreted as only an indication and also the view might choose to set hard limit on maximum number of records retrieved in a data page but if the client request all records (page size set to zero) it will not ask for more data even if the business logic returns less than the total number of records. It is recommended to better throw an error if number of records that are to be returned exceeds the hard limit instead of sending incomplete result set, that way the client is at least aware of the problem and might try to retrieve the records using a lower page size.

`logical hasMoreData ()`

Returns true if there is more data to be retrieved.