How To

Table of Contents

How to Guide

This guide mill make you a Darwino Chef, by giving you recipes to solve commons use cases. As with cooking, new recipes can be invented continuously, so this guide will be updated on a regular basis.

JSON Store

This chapter exposes some JSON Store related 'How To'.

Database queries

Darwino offers two different mechanisms for querying the JSON store: cursor based queries and JSQL. The former is designed upon MongoDB query language, with extensions, while the second is a flavor of SQL adapted to JSON store. Depending on your use case, you may pick one of the other.

When to use each query language

Cursor queries are meant to be simpler and database independent, while JSQL ones are closer to the relational database

Cursor Query

- For queries involving a single database
- For queries with selection formula that cannot be executed by the database (be careful of the performance)
- When executing full text search queries
- When using query fields (@fields) extracted at document saving time
- When retrieving categorized data, with or without aggregated data
- When retrieving hierarchies of documents (parent/children relationship)
- When migrating Notes/Domino view, particularly when using @formula for the columns

JSQL

- For queries involving multiple databases, using joins, union, ... and other relational operations
- For complex calculation with subqueries
- When the full power of SQL is requested for complex queries

Full Text Search

This is actually combined with cursor queries, as a full text expression is a optional parameter of the cursor.

Cursor queries

What are cursor entries

Aquery result can be a set of entries or a set of documents:

- Cursor entry An entry can be either a category or a document. The category contains contains a set of meta-data defining it (isCategory(), getUnid()...) as well as the value extracted from the database.
- Document Every single row is a JSON document matching the query. There is no category in this case

Executing a query in Java

Aquery can be executed using the Java API, by instanciating a Cursor, setting options and executing the query. The result can be either cursor entries, documents or the number of entries/documents being returned. See: https://playground.darwino.com/playground.nsf/JavaSnippets.xsp#snippet=Json_Store_Cursor/

Executing a query using REST services

Queries can be executed using REST services. The result rows are the JSON serialization of the Java objects, having the meta-data at the object root and the value in a `json` property. But, for document services, an optional `/json` is appened, the JSON will just containing the values in its root, without meta-data. if the `DOCUMENT_METADATA` options specified, then the meta-data are added in a property called `__meta`

See: https://playground.darwino.com/playground.nsf/OpenApiExplorer.xsp#openApi=Json_Store_Query/

Passing parameters to queries

Queries can use parameters within the JSON query definition. They are identified by `$$<name>`, similarly to MongoDB. Here is an example of a :brand parameter:

```
Cursor c = store.openCursor()
           .query("{state:'$$state'}")
           .param("state","CA");
```

See: http://playground.darwino.com/playground.nsf/JavaSnippets.xsp#snippet=Json_Store_Cursor_Select_Query_Parameters

Registered queries

If queries can be executed dynamically, they can also be registered by name. Registered queries are retrieved using a factory assigned store factory. Acommon practice is to put the queries in `DARWINO-INF/cursors` as `*.cursor` files, and use the predefined factory like bellow: `setQueryFactory(new DarwinoInfCursorFactory(getClass()));`

See: https://github.com/darwino/darwino-demo/blob/develop/darwino-demo/contacts-react/contacts-react-

shared/src/main/java/com/contacts/app/AppDBBusinessLogic.java

A registered query is loaded in Java by using `load()`. This requests the definition from the extension:
`session.getDatabase("playground").getStore("pinball").openCursor().load("Pinball/Brands");`

See:
https://playground.darwino.com/playground.nsf/JavaSnippets.xsp#snippet=Json_Store_Cursor_Run_Predefined_Query

From REST services, the parameter `name` should be used to reference a predefined query. Note that a list of registered queries is available with `/list`

JSQL

What is JSQL

JSQL is a SQL like query language where basically the table are replace by JSON store names and the table columns by simple JSON paths (`$.x.y.z`). Internally, JSQL is converted to the database native SQL and executed.

More information here: http://blog.riand.com/2017/01/when-sql-meets-nosql-you-get-best-of.html

Executing a JSQL query in Java

A query can be executed using the Java API, by instanciating a JSQL cursor, setting options and executing the query. The result is an array of rows, where each row is a row from the SQL rowset

See: https://playground.darwino.com/playground.nsf/JavaSnippets.xsp#snippet=Json_Store_Jsql/

Executing a JSQL query using REST services

JSQL queries can be executed using REST services.

See: https://playground.darwino.com/playground.nsf/OpenApiExplorer.xsp#openApi=Json_Store_JSQL

Passing parameters to JSQL queries

JSQL queries can use parameters within the SQL. They are identified by `:<name>` and replaced at runtime. Here is an example of a `:brand` parameter:

```
SELECT P.$.name name, P.$.brand brand
    FROM pinball P
    WHERE LOWER(P.$.brand)=LOWER(:brand)
```

See:
https://playground.darwino.com/playground.nsf/JsqlSnippets.xsp#snippet=alexa_Which_Pinballs_Are_Made_By

Registered JSQL queries

If queries can be executed dynamically, they can also be registered by name. Registered queries are retrieved using a factory assigned store factory. A common practice is to put the queries in `DARWINO-INF/jsql` as `*.jsql` files, and use the predefined factory like bellow: setJsqlQueryFactory(new DarwinoInfJsqQueryFactory(getClass()));

See: https://github.com/darwino/darwino-demo/blob/develop/darwino-demo/contacts-react/contacts-react-shared/src/main/java/com/contacts/app/AppDBBusinessLogic.java

A registered query is loaded in Java by using `load()` . This requests the definition from the extension: `JsonArray json = (JsonArray)session.openJsqlCursor().load("MyQuery");`

See:
https://playground.darwino.com/playground.nsf/JavaSnippets.xsp#snippet=Json_Store_Jsql_Run_Predefined_Query

From REST services, the parameter `name` should be used to reference a predefined query. Note that a list of registered queries is available with `/list`

Full Text Search

Darwino cursor queries can be used to execute full text search on the JSON data. Behind the scene, Darwino uses the built-in database full text search capability. On some databases, like MS SQL Server or IBM DB2, the relational databases have to be initialized properly to enable the feature. Failing to do that will generate runtime errors.

How the Full Text Search Works

The whole JSON document, or only a subset of it, can be indexed to support full text search. As most relational databases don't know how to index JSON, Darwino is storing a copy the data to index into a specific table. This has to be defined in the database definition, for each single store:

```
_Store store = db.addStore("mystore");
store.setLabel("My Store");
store.setFtSearchEnabled(true);
_FtSearch ft = store.setFTSearch(new _FtSearch());
ft.setFields("$");
```

In this example, the whole document (`"$"`) is being indexed

Can the Full Text Search be Enhanced

Darwino created an API that works with many different databases, without leveraging every specific database capabilities. But, as this is built on relational databases, one can create specific database indexes and then use them directly with SQL.

Implement Darwino Securty

This chapter exposes some security related 'How To'.

Database Security

Security levels

The Darwino stores uses 3 levels of security:

1. Server level
2. Database level
3. Document level

Server level

Access can be granted or denied for user to the whole database server. This is done by instantiating `ServerACL` object and then assigning to the server extension registry. Without such an object created, all users have access to the server

Database access level

Assuming that a user has access to a server, the user access, and permissions, can be a defined using a `DatabaseACL` object. This object defined privileges from simple readers to managers, while also allowing fine grained control to document creation, update and deletion. The `DatabaseACL` can come from the database definition as one of its properties, or from a factory assigned in the registry. If the registry factory defines a `DatabaseACL` for a particular DB, then it overrides any coming from the database definition. Acommon factory (`DefaultDatabaseACLFactory`) dynamically reads the definition from the object design element, making the security very flexible and dynamic.

Example of a database ACL assigned in the database definition:

```
_DatabaseACL acl = new _DatabaseACL();
acl.addUser("demo", _DatabaseACL.ROLE_READER);
acl.addAuthenticated(_DatabaseACL.ROLE_FULLEDITOR);
db.setACL(acl);
```

Document access level

Each document stored in the database can contain `_readers`, `_writers`, `_ereaders` and `_ewriters` fields. These fields can contains user dn, roles or groups. Even more, they can contain subfields if the entries have to be grouped for easier handling (ex: `_readers.myfield`). As document based security as a cost, it has to be enabled explicitly in the database definition like this:

```
db.setDocumentSecurity(Database.DOCSEC_INCLUDE);
```

If there is no reader security (everyone can read any document), then there is another flag that suppresses the read check, for again performance reasons:

```
db.setDocumentSecurity(Database.DOCSEC_INCLUDE|Database.DOCSEC_NOREADER);
```

Simulating Notes/Domino security model

If Domino provides similar N/D security concepts, there are a few thing to take into consideration:

1. The `DatabaseACL` coming from Domino is, by default, replicated as a design element. But, to take advantage of it, the `DefaultDatabaseACLFactory` must be assigned to the registry. This factory can also be customized to only handle a set of databases.
2. The reader/writer fields should be set to the Notes/Domino compatibility mode as the default Darwino behavior is a bit different.

   db.setDocumentSecurity(Database.DOCSEC_INCLUDE|Database.DOCSEC_NOTESLIKE);

Services Security

Web application server security

If the web application server security is enabled, like in web.xml, then this applies to the all the Darwino services served by this application. Note that this only works on JEE applications, an not on mobile application which should not really matter as only one, local user is being used.

Darwino specific security

Darwino specific security is provided through the `HttpServiceConstraints` class. If this class is very generic, its implementation `HttpConstraintsStatic` providing some easy to use settings is generally used.

An `HttpServiceConstraints` instance can be assigned at different places, depending what the JEE application is using. Most of the time, security is being handled through a JEE filter which can be:

1. Authentication filter If an authentication filter is used instead of the web app server authentication capability.
2. Security filter When the web app server authentication is used, then one can use a security filter that will apply the Darwino security. Note that a security filter should not be created when Darwino authentication is used, as this will be redundant.

In both cases, the security constraints can be assigned explicitly (using setConstraints in the filter constructor), or using a managed bean of type `darwino/httpconstraint`. Even an init parameter, `httpConstraints`, can be defined to use a particular bean name.

For some rare, unusual use cases, HTTP contraints can also be directly assigned to more specific classes, like the service factory or the service dispatcher servlet/filter. In that case though, beans cannot be used and the assignment has to be done explicitly by code.

Advanced JSON database services

Generic JSON database services can also be fined grained controlled, beyond the server/database/document ACL:

1. Database access can be controlled Multiple applications deployed on the same server can share the same database environment. But the services exposed by one application should only give access to the databases used by the application. This is done by default based on the application manifest (see: `J2EEServletJsonStoreServiceFactory`).
2. Filtering data The generic services serve the whole documents or view entries. For security reasons, if some fields should be hidden to some users, then the services have content filters that can be implemented (`DocumentContentFilter` & `CursorContentFilter`).

To filter the content, the AppServiceDispatcher class should create the JsonServiceFactory and the assign it the proper access control or content filters.

Authenticating users

Accessing the authenticated user

Authentication can be done by the web application server, a Darwino filter or the mobile runtime. It is represented by a `User` object containing the dn and other attributes, including its roles and groups.

Darwino application

The easiest way to get the current user in a Darwino application id to use the DarwinoContext. It works both JEE and Mobile apps.

```
    User user = DarwinoContext.get().getUser();
```

Specific to JEE

Although it is advised to use the `DarwinoContent` as this is portable across platforms, a JEE app is has the authenticated user is available through the Authentication Service

```
    User user = Platform.getService(AuthenticationService.class).getUser(request);
```

Extending the directory

Dynamic user attributes

While a directory will set the user attributes, groups and roles, it is also possible to assign roles using a `UserRoleProvider`. It could also be used to add other attributes or groups. Here is, for example, how a static role provider can be assigned to a directory.

```
<bean type="darwino/userdir" name="base" class="...">
   ...
  <bean name="roleProvider" class="com.darwino.config.user.UserRoleStatic">
       <list name="roles">
           <bean class='com.darwino.config.user.UserRoleStatic$Role'>
               <property name='name'>admin</property>
               <list name='users'>
                   <value>user1</value>
                   <value>user1</value>
               </list>
           </bean>
       </list>
   </bean>
   ...
 </bean>
```

Contextual users

The same user can have different attributes, depending on the context. For example, this user can be an admin in a

context and just a user in another one. This is particularly true when instances are used. To support contextual users, the `DarwinoContext` has a `UserContextFactory` that decorates an existing user with contextual attributes/roles/groups, for every single request. An example for that is the IBM `CommunityContext`, which adds roles to user depending on the current community (instance) and the membership of this user in the community. Because this is on a per Darwino request basis assigned to the `DarwinoContext`, the context factory must be defined at the filter level in `web.xml`:

```
<!--  Filter for creating the Darwino Application, Context and DB session -->
<filter>
    <filter-name>DarwinoApplication</filter-name>
    <filter-class>com.darwino.j2ee.application.DarwinoJ2EEFilter</filter-class>
    <init-param>
        <param-name>userContextFactory</param-name>
        <param-value>com.darwino.ibm.connections.usercontext.CommunityContext</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>DarwinoApplication</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```