DBConnect Guide

Release 1.2

MiG InfoCom AB
S:t Olofsgatan 28a
753 32 Uppsala
Sweden

www.miginfocom.com

Table of Contents

MiG Calendar Tutorial

Preface

This document aims at providing enough information to get started using the DBConnect plugin with the MiG Calendar component. The **MiG Calendar Tutorial** contains information on how the component is structured including a overview of its different parts.

The **MiG Calendar Technical Specification (API JavaDoc)** will provide details and should be used as a reference. It can be found at the web site indicated below and should also normally be installed adjacent to this document.

Many IDE:s (Integrated Development Environment) of today have good support for inline help using JavaDocs. The standard HTML JavaDocs for the MiG Calendar component is installed by default and can also be obtained from the site as described below. We highly recommend using this feature as it increases productively when creating applications with this component.

Although all developers independent of prior experience can benefit from reading this document, general knowledge of the standard Java API and OOP (Object Oriented Programming) will help understand some of the details and why they are implemented in a certain way.

Resources and Developer Support

MiG InfoCom AB provides support through email and the online forums. Information and updated tutorials will be made available on the MiG Calendar product site

Contacting Support directly via Email

`support@miginfocom.com`

Submit a support ticket

`http://www.migcalendar.com/support.php`

MiG Calendar Product Site

www.migcalendar.com

Bug Reports

Please submit a support ticket.

How to read this Document

This guide is written to give an understanding on how to use DBConnect. This is not an API reference. You should have the JavaDoc API reference open at the same time as reading this as a reference for method signatures.

There are example source code installed adjacent to this document. It should be used as a reference as well.

Setting up DBConnect for usage

Classpath

In order to use DBConnect your application needs to find it. How to do this depends entirely on you environment but normally you add it to your classpath, possibly in the project settings in your IDE. The file to add to the classpath is `dbconnect.jar`

MiG Calendar Component

In addition to adding the DBConnect to your classpath you also need to setup the MiG Calendar Component for use. How to do this is specified in the Getting Started Guide.

Supported Databases and JDBC Drivers

DBConnect has be developed with, and tested against, the following databases and JDBC drivers:

Microsoft SQL Server 2000 – jTDS 1.2, MS JDBC 2000 and MS JDBC 2005 reference JDBC drivers. jTDS and MS JDBC 2005 driver are highly recommended.

PostgreSQL - Their own provided JDBC driver.

MySQL (4.1 and up) - Their own provided JDBC driver.

IBM DB2 - Their own provided JDBC driver.

HSQLDB 1.8 - Their own provided JDBC driver.

Oracle 9 - Their own provided Thin JDBC driver.

It should be noted that it is the exact same Java code that is run for the different databases*. Standard SQL is used and JDBC functionality that is supported by all databases/drivers above. This means that it is quite probable that other databases is working as well. For adding a database to the supported list, after thorough testing, contact consulting@miginfocom.com.

* Oracle returns all numeric column values as `BigDecimal`. This is why an `OracleAdapter` is included. The code for creating the database, which is normally only done for testing purposes, has slightly different defaults as well.

Overview

Database Updates

DBConnect is monitoring changes in MiG Calendar's `ActivityDepository` and `CategoryDepository` classes and when changes are discovered converts that change to a database action that is put in a queue. There is a separate thread that is reading from this queue and commits the actual changes to the database.

Database Reads

The database is read in the same way it is written. A read action is put on the database queue and when it is its turn, which is normally right away, data are read from the database and put in the corresponding depository in memory.

ActivityDepository and CategoryDepository

MiG Calendar has two central classes that holds activities (e.g. events, todos and tasks) and categories (e.g. priorities, activity owner(s), basically any way you would like to “tag” activities) respectively. This makes managing entities (Activities and Categories) simple for the application developer. These two classes are also the focal point for DBConnect. The advantage with this approach is that since these two classes are used as the in-memory storage for entities in all MiG Calendar applications, there is little, if any, changes that are needed to make the (your) application synchronization aware.

The DBConnect Architecture

At the lowest level a property of the activity/category is mapped to to a column or an XML element/attribute with a `PropertyMapping` object. All property mappings that are intended for one database table or XML element are then

collected into one `PropertyMappingList` object. To that `PropertyMappingList` object are optionally also a special update table associated that has its own, very simple, `UpdateMapping`. One can also connect a number if mappings to other tables in the form of `IDToManyPropertyMapping`s. These are for instance to map a single Activity to several Categories though the use of a join table. Both of these features are explained further below.

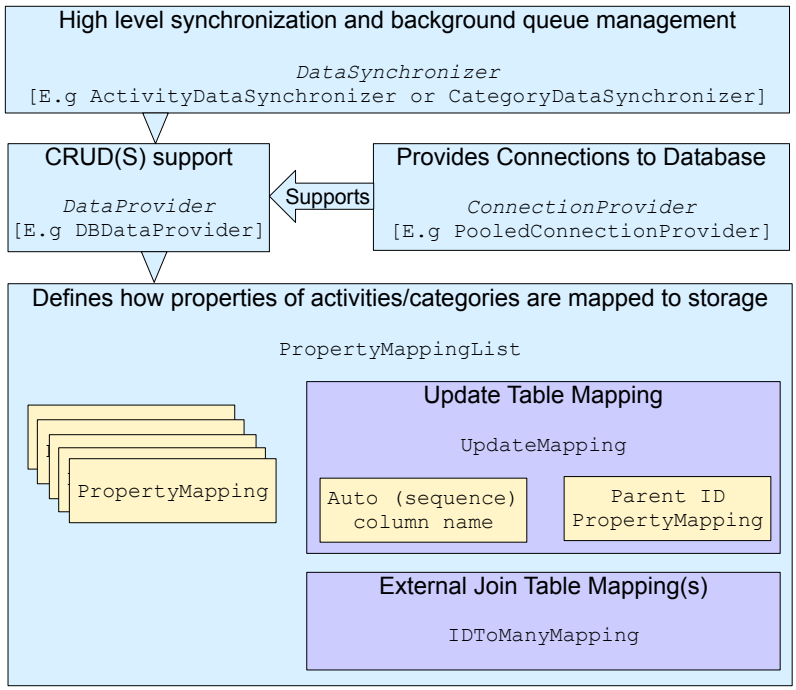
A `DataProvider` is wrapped around the `PropertyMappingList` object and, for use with a database, a `ConnectionProvider` are included as well. The `DataProvider` has all the information on how to read/write properties from the activities/categories and load/save it to a backing store. For database persistence a `DBDataProvider` is used and you can for instance use an `XMLDataProvider` to read/write from an XML file (XML implementation not provided with 1.0).

We must also add the more high level functionality of monitoring the `ActivityDepository` and `CategoryDepository` as well as provide background processing in a worker thread/queue. This functionality is provided by the `DataSynchronizer` which normally is an `ActivityDataSynchronizer` or a `CategoryDataSynchronizer`.

From that synchronizer you control almost every aspect of the DBConnect plugin.

Layered Architecture

DBConnect is created in a layered architecture. You can choose just to configure it or to add you own persistence delegates instead of the ones provided by the plugin.



The normal object hierarchy for both categories and activities.

Multiple Databases

You can have more than one DBConnect instance running at the same time, they can also handle the same activities or categories and be connected to different databases or XML files. Since the protocol used in DBConnect is versioned the there shouldn't be any problems with finding the newest version of an entity.

Problems can arise when the same entity (e.g. activity) are changed in two databases without any synchronization in between. The `LastModified` property can correctly be used when the change is only made in one database assuming that the last modified field is set correctly.

All comparisons are made when an entity is loaded and it is then compared to the version of that entity already in memory. There is currently no way to use DBConnect to synchronize two databases/backing stores in a more generic way. This is something that will possibly be added in a release after 1.0.

A good way to architect the possibility to connect to one database that is not guaranteed to be online 100% of the time but still use the application when it is *not* online is to use a local database as an extra DBConnect backing store. For instance

HSQLDB 1.8 is a viable choice for this and it is also one of the supported database engines, as well as free to use.

DBConnect is actually not the one that detects if more than one version of an entity exists. The depositories (ActivityDepository and CategoryDepository) already handles this. You can install a *resolver* to handle these cases. See the JavaDoc for `ActivityDepository.setActivityResolver(..)` and `CategoryDepository.setCategoryResolver(..)` for information how to install your custom resolver.

Multiple Clients

If there are more than one client reading from and writing to a backing store there is a chance that two clients update the same entity at the same time. DBConnect does something called optimistic locking which detects such attempts but do not hinder them altogether as true locking would (true locking would require that all databases was online at all times though). See "Listening for Concurrent Events" below for how to handle this.

Setting Up the Property Mappings

What is a Property?

Activities and Categories both implement the interface `PropertyConsumer` which means they can both be used to read and write properties from/to. The type of the property may be any Java object. The property is defined with the class `PropertyKey` which is, as the name indicates, used as a key to get/set the property. Property keys has a string name and an optional enforced value type, such as `Integer`. They are architectured so they are extremely fast to use for lookup of the property since referential equality (==) are used.

Both activities and categories have standard properties such as *id*, *date range* and *name*. You can also freely have your own properties and store those in the activity through the use of `PropertyKey` objects as keys and just about anything as values. This way you can extend the functionality of the `DefaultActivity` class without sub classing it.

Property Mappings

To be able to load and save (persist) an entity to a database or XML file you need to map the properties of the activity (or category) to

corresponding columns in a database or elements in an XML document. This mapping is almost the only thing you need to tell the DBConnect plugin for it to work. Some properties need, for instance to optimize database queries, to be scattered over one or many database columns. The opposite is also true. Some database column values is calculated from more than one property of the activity. For instance the “affected start and end date/time” column are calculated from the base date range of the activity together with an optional recurrence rule.

Load/Save Mappings to File

All classes that are used to map DBConnect to a database structure are both `Serializable` and have XML delegates so they can be loaded/saved to XML. There are convenience load and save methods in `DBUtil` that handle the specifics.

Mandatory Properties

Some properties of the activity and category are mandatory to store in the backing store (database or XML file for instance). Take the “affected start and end date/time” above. The plugin must know which is the first and last date/time that the activity will affect so that it doesn't have to load all activities if it just wants to show one day in the calendar.

Those mandatory property mappings normally have `static` convenience factory methods in the `PropertyMapping` class but they also exist as public classes.

Here follows a complete list of the mandatory mappings. Note that there are no arguments for the methods. Pleas see the example code and/or JavaDoc API Documentation for this.

For *Activities*:

ID – This is the ID of the activity. Normally an `Integer`, `Long` or `String`.

To create the mapping use:

```
new IDMapping(...)
```

Affect Start/End - This is the total date range that the activity will affect expressed in milliseconds. In this value the recurrence rule is included and thus the end can be set to an extremely high value if the activity is recurrent without a stop date. The database column must return a `Long`. This value is typically used to optimize the database

query.

To create the mapping use:

```
new ActivityAffectedMapping(...)
```

DateRange – This is the normal date range that the activity spans. In case of a recurrent activity this is the base date range. The database column must be a string that can hold at least 64 characters.

To create the mapping use:

```
new DateRangeMapping(...)
```

Recurrence – If the activity is recurrent this is where the definition is stored. The value is an XML text which has a length proportional to the complexity of the recurrence. Since recurrence rules can be combined into very complex recurrence rules with MiG Calendar there is no upper limit to how long the text string can be. Normally about a hundred characters. But none if the activity is not recurrent.

To create the mapping use:

```
new XMLDelegateMapping(...)
```

Categories – There are two ways to persist which categories the activity is “member of”. One where a comma separated string representation of the connected category id:s is stored as a value and one where a separate many-to-many table connects activity id:s to category id:s. The former is simpler and faster, the latter is more database correct and more flexible in some cases.

The *single column approach* is created like any other property mapping. You will have to provide the class type that the category id has and a delimiter (e.g. a comma sign) that is guaranteed not to be in the string representation of the id itself. The class type that is used as the id for the categories must have a constructor that takes a single `String` argument. For instance `String`, `Integer` and `Long` has this already. The column type in the database must be string of a length that can hold as many characters as there will be category id:s for a single activity expressed as a comma separated string.

To create the mapping with this approach use:

```
new StringArrayMapping(...)
```

The *id-to-many join table* mapping approach is from a database design perspective the better choice since it avoids having references to other entities in a composite database column. There are no special demands on the object used as category id other that it must be able to be mapped to a singe column.

The id-to-many join table is normally a simple table with two columns, each holding an id to another object (activity/category). This way one can express which categories belongs to which activities in a very flexible manner. The disadvantage with this is that the retrieval of these relations has to be done with a second query to the database, which is slightly slower. The advantage is that the you can with an SQL query directly for instance ask for “all activities that is connected to a certain category”. This can't be done if the category id:s are embedded in one column as in the singe column approach above.

To create the mapping with this approach create the appropriate `IDToManyPropertyMapping` objects and set them on the `PropertyMappingList`.

Last Modified – The last modified property should be represented as a `Long` (bigint) in the database. To create the mapping with this approach use:

```
new LastModifiedMapping(...)
```

Status – This is the status of the record. Currently there are only two values. `0` or `null` means normal status and `1` means deleted. The reason deleted records aren't simply removed from the database is that if one user deletes it and another user's DBConnect detect its not in the database it will re-create it rather than delete the local version.

The status is only exiting in the database and in the internals of DBConnect. It will not be saved in the activity itself when loaded into memory.

To create the mapping for this property use:

```
new StatusMapping(...)
```

Version – Every activity in the database is tagged with a version number that is changed every time the activity is changed. This gives DBConnect the possibility to update the database from more than one source and still not lose information. The version is an `Integer` and should be set to that type in the database. The version column can be omitted to increase performance but if so the access to the database must be single user (only one client).

The version is only exiting in the database and in the internals of DBConnect. It will not be saved in the activity itself when loaded into memory.

To create the mapping for this property use:

```
new VersionMapping(...)
```

For *Categories*:

Parents – Since categories are hierarchical this relationship must be persisted to the database. The actual relationship is much like for the activities' Category mapping above only that here it points to other category id:s. Instead of repeating the text here we refer to the explanation for Categories above. It should be noted that there is much less need for the join table for this property since the number of categories are usually much smaller and the are usually all read into memory all at once. They are also updated much less frequently.

Categories can have more than one parent. This is only for special cases where this is needed and normally a category will only have one parent, except for the root which has a null id and are not persisted.

ID, Last Modified, Status and Version - These are mandatory columns also for the categories. They are exactly the same as for activities so they are not re-iterated here.

Custom and Non-mandatory Properties

Both activities and categories support the addition of custom properties. You can for instance add a property to activities that states if they are disabled or not. Chances are that you want to include this property in the data that is persisted so that it is preserved from one run to another.

You handle these custom properties like any property on the activities/categories, by setting up a property mapping for it. You will actually do this for the non-mandatory properties *summary*, *description* and *location* anyway. `SimplePropertyMapping` has been made for this and you create one of these as you create any object in Java:

```
new SimplePropertyMapping(name, propertyKey,
mapType, colType);
```

Refer to the sample code provided with the plugin for examples of the usage of these properties.

Update Tables for Increased Performance

To increase performance in a multi user environment (multiple DBConnect clients are connected to a database at the same time) DBConnect support the use of update tables. These tables, one for activities and one for categories, contains rows with an id of which

entities have been changed and one auto increase column that just increases it's value for every new record.

This gives DBConnect the option to, in a very fast and resource slim manner, ask the database which records has been created or updated since change X. DBConnect handles everything automatically if it is just fed the names and definition of those tables. See the example code for how to do this.

To enable this feature create a `UpdateMapping` object and set it on the `PropertyMappingList`. See `DBUtil.createUpdateTable(..)` for how these table should be created.

Database Connection

DBConnect must be told how to connect to the database that it should use for persisting activities and categories. This is done in a standard JDBC manner with maximum of flexibility. The `ConnectionProvider` interface is used for the purpose. It is very simple and only defines how to get/return a connection and a callback method for when the provider isn't needed any more (.dispose()).

There is also a concrete implementation that will be sufficient for most cases. It has connection pooling, pre- and/or post connection verification with customizable SQL as well as maximum live connection count. The implementation is in the class `PooledConnectionProvider` and how to use it can be view in the example code.

Example Database Mapping

For versioning purposes the code is not pasted into this document. During the installation of DBConnect the demo code installed. You can view this code and use it as a reference.

Using MiG Calendar the Correct Way

Event Types

When you add or remove activities/categories from their respective depository the event type that should be fired is passed along as an argument. The reason for this that DBConnect must know if the activity/category was removed because it wasn't needed in memory anymore or if it was deleted by the user.

There are four event types and DBConnect reacts appropriately on

these events, making it a snap for you as a developer to use DBConnect without almost any changes to the code. The events are defined in the class `TimeSpanListEvent` for activities and `CategoryStructureEvent` for categories. They have the same event types defined and are interpreted as follows:

`CategoryStructureEvent/TimeSpanListEvent.ADDED`

One or more activities/categories was added but usually not as a result of a direct creation. It might be added from another source where it was created for instance.

`CategoryStructureEvent/TimeSpanListEvent.REMOVED`

One or more activities/categories was removed but usually not as a result of a direct deletion. If might be removed because it no longer was needed in memory for instance.

`CategoryStructureEvent/TimeSpanListEvent.EXCHANGED`

An Activity or Category was exchanged for another one.

`CategoryStructureEvent/TimeSpanListEvent.ADDED_CREATED`

One or more activities/categories was added as a result of creation. The creation part might for instance hint synchronizing tools that they should add this to their database.

`CategoryStructureEvent/TimeSpanListEvent.REMOVED_DELETED`

One or more activities/categories was removed as a result of deletion. The deletion part might for instance hint synchronizing tools that they should remove this from their database.

Miscellaneous Topics

Logging

An important tool for troubleshooting and event tracking is the logging build in to DBConnect. You can set any Logger object on any of `CategoryDepository`, `ActivityDepository`, `DataSynchronizer`(s), `ConnectionProvider`(s) or `DataProvider(s)`. This includes subclasses as well of course.

Almost any action in DBConnect and the depositories (which is actually a part of the main MiG Calendar Component) are logged at

some level. You can set the level on the Logger to get everything from errors to fine grained details of what is happening. For troubleshooting it is almost a must to use logging since the DBConnect is using a background thread to do the database operations.

There is a method `setLogger(Logger l)` on all of the above objects types.

For information on Logging see the documentation for Java 1.4+ API at java.sun.com. E.g.
http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/

Creating a Database from the Mappings

Primary for testing and debugging DBConnect has utility methods for creating a database (or more correct the *tables*) according to the current property mappings. The database itself isn't created, nor are the database users.

`DBUtil.createTable(con, tableDef, dropFirst, constaints)` will create the table.

`DBUtil.createUpdateTable(con, tableDef, dropFirst, constaints)` will create the update table, if any.

`DBUtil.createIDToManyTables(con, tableDef, dropFirst, constaints)` will create the any external Join tables.

Constraints on the tables, such as primary keys, are not automatically handled, but can be sent into the method as an optional argument and will be shunted to the database. For instance the standard SQL phrase `"PRIMARY KEY (activity_id, category_id)"` can be used to specify that those two columns should be part of the primary key for an `IDToManyTable`.

Note that the create tables method returns the SQL as a `String`. If you set the connection to `null` the database will not be contacted, and thus no tables created, but you can get the SQL string. The SQL can then be used as a reference on how DBConnect expects the database to look like. It can also be used to send the code manually to the database with a tool provided by the database vendor.

See the demo code for examples.

Deleting Table Data

There is also a utility method for delete all data in a table. To delete all rows in a table use:

`DBUtil.deleteDataInTable(connection, tableName)`

Monitoring the Database Queue

DBConnect uses a background thread to communicate with the database. This ensures that there will be no lags or lockups of the user interface when information is persisted to the database. There are multiple ways you as the developer can interact with this queue. You can check the length of the queue, wait for it to flush, shut it down as well as check its contents.

`DataSynchronizer` is in many ways the focal point of DBConnect while it is running. Through that class you can monitor the database queue including starting and stopping it.

You can also add a listener on the database queue that will be notified every time the queue count is changed.

View the `DataSynchronizer` JavaDoc for details on how to use this class.

Filtering What to Synchronize

Since you might want to have activities/categories in your memory, and thus the depositories, that you want to exclude from DBConnect you can create and set a `CRUDSFilter` that filters which object to do Create, Read, Update, Delete and Synchronize on respectively. You create the filter by sub classing `CRUDSFilter` and implement the single abstract method `accept(Object o, int operation)`.

View the `CRUDSFilter` JavaDoc for details on how to use this class.

Polling and Setting Auto-poll Interval

The `DataSynchronizer` can check the database for changes (polling) automatically every X milliseconds.

View the `DataSynchronizer` JavaDoc for details on how to use this class.

Auto Write

This is a property of the `DataSynchronizer`. It is used to turn on and off the automatic monitoring of changes to activities/categories. If this property is set to `false` DBConnect will not pick up changes and they will thus not be handled and persisted.

View the `DataSynchronizer` JavaDoc for details on how to use this class.

Managing Shutdown and Flushing

You can flush the database queue and wait for it to finish, optionally a

maximum of X milliseconds, or you can showdown the queue after it has reached size 0.

View the `DataSynchronizer` JavaDoc for details on how to use this class.

Queue Action Coalescing

This is an advanced and unique feature of DBConnect. If two database actions will counter each other out, for instance if the summary is set in two actions after each other, they can be combined into one database action. Another example is two poll actions that are in sequence without delay between them can for instance be merged into one.

The database coalescing can be set in currently three modes:

`DataSynchronizer.COALESCE_NONE` - means that there will be no coalescing.

`DataSynchronizer.COALESCE_ADJACENT` - means that two data action will be coalesced only if they are next to each other in the queue.

`DataSynchronizer.COALESCE_REORDER` - means that two data actions will be coalesced so that the data action earlier in the queue will be removed if a later addition will counter that action out, even if there are other actions between them. This will in effect sometimes reorder data actions and they might not end up in the same order at the backing store as they where produced. Normally this is not a problem though.

You set the coalescing mode on the DataSynchronizer, for instance the `ActivityDataSynchronizer`.

`COALESCE_ADJACENT` is the default mode.

Pausing and Resuming the Database Queue/synchronizer

The `DataSynchronizer` (and thus the `ActivitySynchronizer` and `CategorySynchronizer`) has a `setPaused(boolean)` method. It can be used to pause the data queue as soon as the current data action (if any) is processed. When the synchronizer is resumed the pending data actions will continue.

Note that the data queue will still be added to even if the synchronizer (which owns the queue) is paused. If `AutoWrite` is `true` there will be create/update actions added when activities and/or categories are changed and if the

`pollIntervall` is `> 0` poll actions will be added, though coalesced (removed) if next to each other in the queue.

Adapting to Other Databases (DBTypeAdapter)

For instance the JDBC driver for the Oracle database returns a `BigDecimal` object for `Integer` and `Long` columns. Other untested databases or JDBC drivers may have similar oddities and the `DBTypeAdapter` are created to accommodate for these cases.

`DBTypeAdapter` is a converter that converts objects returned from the JDBC driver into more appropriate Java object types that can be handled by DBConnect. You install the converter on a `DBDataProvider` instance.

View the `DBTypeAdapter` JavaDoc for details on how to use this class.

Listening for Concurrent Update Events

Since DBConnect is using the high performance *optimistic locking* algorithm there are the possibility that if data are updated from two clients at the same time one of the updates will be lost. Optimistic locking will identify that the data **will be** lost though and you can listen for these occasions by adding a `ConcurrentUpdateListener` to the `DataProvider`. The listener can with customizable logic decide what to do in these cases. Re-read the data or force an update to the database anyway, even though the data in the database has been changed since it was last read.

View the `DataProvider.addConcurrentUpdateListener` JavaDoc for details on how to do this.

Listening for Exceptions

Since DBConnect is using a background thread to communicate with the database it is not possible to surround the actions that will lead to a database interaction with a try-catch block. You can get the same result by adding an `ExceptionListener` to the `DataSynchronizer`. It will be notified if a database action resulted in an exception, including a reference to the original exception.

If no listeners are registered with the data synchronizer the exception will be dumped to `System.err`.

View the `DataSynchronizer.addExceptionListener` JavaDoc for details on how to do this.

Note that if a `DataAction` (for instance a poll or save) is retired more than `retryCount` number of times it will be put back first in

the queue and an exception is fired to the `ExceptionListener`s. It is then up to the listeners to handle the situation.

Loading only Visible Activities

The DBConnect activity poll does not out of the box know about what date/time ranges the user are currently watching in the date areas. This means that it will load ***all*** activities which is many times not desirable, since keeping them all in memory can be too much. Even though MiG Calendar can typically hold about 100.000 activities in a normal PC's memory it will still reduce performance.

There is an easy way you can make DBConnect only load activities that are visible. You can even add date ranges that are interesting other than the ones handled automatically by the date areas. All date areas (e.g. `DefaultDateArea`) register their visible date range as a `Subscription` to the `ActivityDepository`. What you need to do is to add a specific `WHERE` clause to the `DBDataProvider` that tells the database to only return activities between certain ranges. Fortunately there is a utility method that creates this `SQLExpression` for you and it is even dynamic in that it reads the active subscriptions for every call.

All you have to do is:

```
SQLExpression expr =
    DBUtil.getDynamicActivitySubscriptionsSQLExpression("affect_start_millis",
                                                        "affect_end_millis", null);

DBDataProvider provider = (DBDataProvider) activitySynchronizer.getDataProvider()
provider.setReadWhereExpression(expr);
```

There is also a static version of the method above. It will create a filter for the subscriptions as they are registered at the time of the method call.

Custom Properties in Activities or Categories

Both the `Activity` and `Category` classes uses a `PropertyKey` to get and set property values. This is happening in the background even if you use the convenience methods such as `activity.setSummary("summary")` which is the same thing as calling `activity.setPropertySilent(AbstractActivity.PROP_SUMMARY, "summary")`.

This makes it very easy to have custom properties in your activities and categories. All you have to do is create the `PropertyKey` by

calling `PropertyKey.getKey("myPropertyName")`. The key returned can be used to store any property in any `PropertyConsumer` (which both the `Activity` and `Category` classes implement).

When you have stored your value(s) they are as easy as any built in property to persist. You only have to create a property mapping as explained above. If the value yo want to store is of a type that is unknown to DCConnect, such as your own defined classes, those can be persisted as well and this is explained in the next section.

Persisting your custom data types

To persist a property in the Activity or Category that has a class type that is a bit more complex than for instance `String`, `Integer` or `Long` you will have to create your own `PropertyMapping` sub class. It is used to convert to and from the custom class type and a type that is recognized by the database, which normally means `String`, `Integer` or `Long`.

There are two ways to go about this. Use a `String` to store the object in the backing store or to map the custom type to two or more fields in the backing store (e.g. columns in a database).

Note! *It is currently not possible for an Activity or Category to have the ID object stored in more than one database column. The ID* ***can*** *be a custom class type but it has to be stored in a single field (column) in the database. Put in another way, the the primary key of the record can not be more than one column.*

The abstract methods in `PropertyMapping` are easy to understand how to implement given the API JavaDoc. Here is a straight forward implementation of a property mapping that writes and reads a type `MyComplexType` that holds two integers.

```
public static class MyComplexPropertyMapping extends PropertyMapping
{
    private static final PropertyKey MY_KEY = PropertyKey.getKey("myComplexKeyName");

    public MyComplexPropertyMapping(String columnName)
    {
        super(columnName,
              PropertyMapping.MTYPE_GENERIC,
              PropertyMapping.STRING_NOT_NULL);
    }

    public void setPropertyValueFromStorage(Object o, PropertyConsumer pc)
    {
        String[] parts = ((String) o).split(",");
        MyComplexType value = new MyComplexType(parts[0], parts[1]);

        pc.setPropertySilent(MY_KEY, value, Boolean.FALSE);
    }

    public Object getColumnValueToStorage(PropertyConsumer pc)
    {
        MyComplexType value = (MyComplexType) pc.getProperty(MY_KEY);
        return value.getFirstInt() + "," + value.getSecondInt();
    }

    public boolean isAffectedByProperty(PropertyKey key)
    {
        return key == MY_KEY;
    }

    public boolean canConvertFromStorage() { return true; }
    public boolean canConvertToStorage() { return true; }
}
```

There is also the possibility that you want the complex type to be persisted to two or more database columns, for instance one column for each of the Integers in `MyComplexType` above. This is more correct from a database optimization perspective and opens up for SQL queries on these columns, something not suitable if you are using the composite approach as describe in the code above.

Setting up the property mappings for storing the `MyComplexType` to the backing store is not very hard. Just use one `SimplePropertyMapping` that writes from the real key to the backing store but reads it back into two temporary properties. The override is so that the value that are saved is the correct part of the complex type.

```
// Keys normally created somewhere else
static final PropertyKey MY_KEY = PropertyKey.getKey("myKey");
static final PropertyKey MY_KEY_TMP1 = PropertyKey.getKey("myKey_tmp1");
static final PropertyKey MY_KEY_TMP2 = PropertyKey.getKey("myKey_tmp2");

// The mappings that goes in the PropertyMappingList
new SimplePropertyMapping("my_col_1", MY_KEY, MY_KEY_TMP1,
             PropertyMapping.MTYPE_GENERIC, PropertyMapping.INTEGER_NOT_NULL ) {
    public Object getColumnValueToStorage(PropertyConsumer pc)
    {
        return ((MyComplexType) pc.getProperty(MY_KEY)).getFirstInt();
    }
},

new SimplePropertyMapping("my_col_2", MY_KEY, MY_KEY_TMP2,
             PropertyMapping.MTYPE_GENERIC, PropertyMapping.INTEGER_NOT_NULL) {
    public Object getColumnValueToStorage(PropertyConsumer pc)
    {
        return ((MyComplexType) pc.getProperty(MY_KEY)).getSecondInt();
    }
},
```

The reason the property can't be read back into the correct property at once is that the mapping is for **one** field in the backing store and when read you only have one of the Integers at a time. This makes it impossible to create the object from one `PropertyMapping`.

To create the final property you add a `Resolver` into the `PropertyMappingList`. `Resolver.resolve(PropertyConsumer)` is called when all properties has been read into the `PropertyConsumer`. All you have to do is to remove the temporary properties your property mappings created and create the real property with them. Something like this:

```
class MyComplexResolver implements Resolver
{
    private static final PropertyKey MY_KEY = PropertyKey.getKey("myKey");
    private static final PropertyKey MY_KEY_TMP1 = PropertyKey.getKey("myKey_tmp1");
    private static final PropertyKey MY_KEY_TMP2 = PropertyKey.getKey("myKey_tmp2");

    public void resolve(PropertyConsumer pc)
    {
        Integer int1 = (Integer) pc.removePropertySilent(MY_KEY_TMP1, Boolean.FALSE);
        Integer int2 = (Integer) pc.removePropertySilent(MY_KEY_TMP1, Boolean.FALSE);
        if (int1 != null && int2 != null)
            pc.setPropertySilent(MY_KEY, new MyComplexType(int1, int2), Boolean.FALSE);
    }
}
```

Then just add that resolver to the `PropertyMappingList` that you added the `SimplePropertyMapping`s you created.

```
PropertyMappingList.addReadResolver(new MyComplexResolver());
```

Troubleshooting

The best way to troubleshoot anything regarding the `CategoryDepository`, `ActivityDepository`, `DataSynchronizer`(s), `ConnectionProvider`(s) or `DataProvider`(s) is to turn on logging for them. That way you can log anything that happens and in what order. See Logging above for more details.

Also check the forums for the DBConnect plugin, the question you are wondering might have been answered there already.

DBConnect misses read-backs (polls):

To be able to support multiple simultaneous databases the synchronizer keeps a record of which entities has been locally deleted and it will ignore reads/updates of those entities. Use DataSynchronizer.reset() to reset the synchronizer to a newly created state where it will read from the database without remembering what has previously been deleted.

General Suggestions and Tips

Some of the more hard to find problems is because the wrong class type for the ID is used. For instance a String with “100” is not equal to an Integer 100. The tip is to use the `setEnforceIDClass(Class c)` in `CategoryDepository` and `ActivityDepository`. It will force both depositories to report any ID object of classes that isn't of the indicated type.