The Servoy Beginner's Handbook

by

Adrian McGilly Information Systems Consultant

McGilly Information Systems, Inc.
Servoy Training & Consulting
San Francisco, CA, USA

Website: www.mcgilly.com
E-mail: info@mcgilly.com
Tel.: 510-428-1035

The Servoy Beginner's Handbook

by

Adrian McGilly Information Systems Consultant

McGilly Information Systems, Inc.
Servoy Training & Consulting
San Francisco, CA, USA

Website: www.mcgilly.com
E-mail: info@mcgilly.com
Tel.: 510-428-1035

Edition: 1.5.2.2

The Servoy Beginner's Handbook

by

Adrian McGilly Information Systems Consultant

Servoy Training & Consulting
San Francisco, CA, USA
Website: www.mcgilly.com
E-mail: info@mcgilly.com

v 1.5.2.2, last updated October 2009

Table of Contents

Throughout the handbook, highlighted section titles refer to sections that are new since the previous edition. Individual passages and paragraphs that have been revised are also highlighted

Introduction

Servoy is a brilliant development environment that raises developer productivity to new heights. It leverages today's most powerful technology standards (Java, JavaScript, JDBC, RMI, AJAX, SQL Databases, application server technology, browser technology, CSS, XML, etc.) elegantly and efficiently in a way that empowers developers like never before.

My name is Adrian McGilly and I am a Servoy consultant and trainer, and a member of the Servoy Alliance Network (SAN). I have a bachelor's degree in Computer Science and Mathematics and I taught myself Servoy in 2006 after a decade of developing database applications, primarily using Omnis and SQL databases. I provide Servoy development, training and one-on-one coaching. For information about online training classes and other services please visit my website.

As I was teaching myself Servoy, I noticed that there were topics that the existing Servoy documentation didn't address fully, and that is why I wrote this handbook. This handbook is a collection of 'how-to' lessons and discussions for developers who want to learn Servoy. It is not a replacement for the Servoy documentation but a complement, one that will help round out your understanding of Servoy and get you up to speed faster.

Feel free to email me anytime at feedback@mcgilly.com with questions, comments, criticism or corrections regarding any aspect of the handbook.

Everything mentioned in this handbook applies to Servoy v. 4.x, unless otherwise noted.

My thanks to John Allen, Marc Norman, Jan Aleman and Bob Cusick who all contributed ideas, corrections and encouragement towards the creation of this handbook. Also a big thank you to all the Servoy developers who ask and answer questions on the Servoy Forum, as those exchanges help me identify what topics to include in this handbook.

by Adrian McGilly, Servoy Consultant & Trainer

How to Learn Servoy

by Adrian McGilly

Available Resources

Because Servoy leverages many different languages & technologies (SQL, SQL Databases, Java, Javascript, HTML, etc.) becoming successful with Servoy requires that you become somewhat familiar in all of these areas.

If you haven't yet worked with a truly relational SQL database, then you'll need to read up on that a bit so that you become familiar with relational database concerpts and know how to design an efficient, normalized datamodel.

- If you don't know HTML at all then read up on it a bit.
- If you don't know what a Java Virtual Machine is then you need to educate yourself a bit about Java.
- If you're new to JavaScript, look at the JavaScript Primer in the *Servoy Advanced Programming Guide for FileMaker Developers*. You don't need to know much JavaScript to get started in Servoy so just get a feel for the basics at first - (grammar, syntax, variable declaration & typing, flow control). I also recommend you buy O'Reilly's "JavaScript, The Definitive Guide" (it's the one with a rhinoceros on it), as the O'Reilly books have always impressed me as being, clear, concise and well-organized manuals.
- If you're new to SQL, take a quick look at the SQL primer in the *Servoy Advanced Programming Guide for FileMaker Developers*. Depending on the complexity of your project, you may never need to use SQL, but it's a good idea to have a basic understanding of what it is and how it is used.

When it comes to learning Servoy itself, there are tons of resources available to help you. Below is a list of all the ones I know of at this time:

- Documentation from Servoy, available for free download as a PDF from the PDF Documentation page of the Servoy website or for purchase in book format from the Servoy Store. This includes:

    Servoy Developer User's Guide

Servoy Developer Reference Guide

Advanced Programming Guide for FileMaker Developers

Servoy Server Administrator's Guide

- The Servoy Help system built into Servoy Developer contains everything that you would find in the Servoy Developer Reference Guide.
- There is some documentation built into the Servoy Script Editor itself. See this section for more information.
- The Servoy Magazine, an online, interactive magazine full of great tips, techniques and how-to's (plus the odd laugh).
- Online Tutorials on the Servoy website
- Virtual Users Group meetings and Webinars put on periodically by Servoy – watch the Events page of the Servoy website
- Online Tutorials on Scott Butler's (The Servoy Guy) website
- Various resources on Greg Pierce's (Agile Tortoise) website
- The Servoy Gallery - a section of the Servoy website that shows screenshots and descriptions of over a dozen Servoy solutions in production. Looking at these screenshots is a great way to see the full range of capabilities of Servoy' GUI.
- Demo solutions which you can download from the Servoy website, explore and use as templates for your own solutions.
- This handbook (*The Servoy Beginner's Handbook* by Adrian McGilly).
- Servoy Training classes, offered online and in person by myself, Servoy, ClickWare Inc. and others
- The Servoy Forum - an online discussion group where close to 1000 developers are asking questions, sharing ideas and getting answers. Development staff at Servoy regularly respond to questions on this forum.

- ServoyWorld, the annual conference hosted by Servoy, is a great place to meet developers and Servoy staff, learn what others are doing with Servoy and find out what's on the horizon from Servoy's engineering team.

- And by far the most important learning tool of all: *Servoy itself*. Play with it. Mess with it. Don't worry – it won't bite. The best way to learn is by doing.

Suggested Learning Path

Here is my suggested learning path for newcomers to Servoy.

1. Get Servoy up and running on your computer, using the Sybase SQL Anywhere database as your database. If you have trouble getting connected to the Sybase SQL Anywhere database read the section about connecting to the database.

2. Watch the Online Tutorials on the Servoy website.

3. Read the Developer User's Guide cover to cover. It's OK to skim parts that don't feel terribly relevant to your particular needs but you should spend some time in every section of this book so you understand the full range of Servoy's capabilities. You should also 'play along' and try things in Servoy as you read.

4. Skim the Developer Reference Guide, but don't try to understand everything in there at first. That will come with time.

5. Open up the svyCRM Demo solution which is provided with the Servoy install and which is available for download from the Download Demo Solutions section of the Servoy Website. Poke around in this solution, both in run-time and design modes. Look at how the forms are put together. Look at some methods. Explore the tables. Don't worry about breaking the demo - you can always reinstall it.

6. If you're a FileMaker Developer, you will find other parts of the *Servoy Advanced Programming Guide for FileMaker Developers* very useful too.

7. You probably have a project in mind that you're aching to try in Servoy. Think of some simple part of that project - an interface for entering and/or browsing Client records, for example, and start building it. As you stub your toe (and brain) against obstacles, consult this handbook - hopefully a good

many of your questions will be answered here. If not, run a search in the *Servoy Magazine* to see if any articles have been written on the topic you're stuck on, and check the *Servoy Forum*. The next section discusses how to get the most out of the *Servoy Forum*.

8. Lastly, at some point remember to take a look at the *Servoy Server Administrator's Guide*. Even if you're still a ways off from doing any server administration, it will help you to understand Servoy if you understand some of the server admin capabilities.

Do I Have To Know SQL To Use Servoy?

It's useful to know a little SQL, but in most cases it's not necessary, and certainly not when you're just getting started with Servoy.

Servoy provides its own set of straightforward commands for retrieving data, but Servoy also gives you the option to query your db using SQL. For example you can pass an SQL query to a function called loadRecords() to retrieve rows from a database and display it on a form exactly as if you'd used Servoy's built-in search commands. This is a convenient option for those who are comfortable writing SQL queries, but you can almost always get by without it.

Certain complex, multi-join queries are beyond the capabilities of Servoy's built-in search functions and can only be achieved using SQL. With a mature, complicated database, directly querying the database using SQL will often improve the performance significantly on some queries.

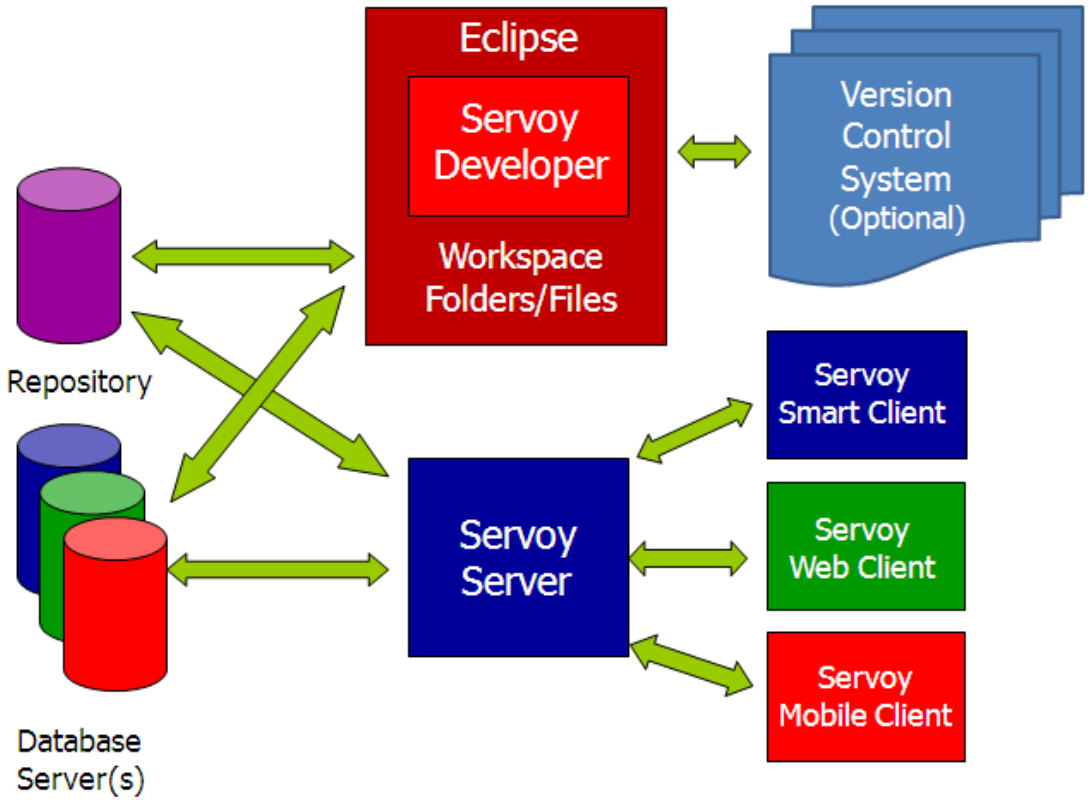
Technically, you can even use the rawSQL plugin to modify your database using SQL update, insert and delete commands, but this is only for expert Servoy users who understand the risks of doing this. Rarely is this going to be necessary.

by Adrian McGilly, Servoy Consultant & Trainer

Overview of Servoy's Architecture

by Adrian McGilly

It's important to understand the architecture of Servoy. Below is a picture of the Servoy Architecture (provided by Servoy)



Applications written in Servoy are called *solutions*.

The first thing to understand is that your solution and your business data are two separate entities in the Servoy architecture (as they should be!). Your data is held in one or more SQL databases managed by one or more database servers. Your deployed solution is also stored in a SQL database called the Servoy Repository. One of the great feats of Servoy is that it stores and manages your entire solution, including your GUI and all your business logic, in an SQL database. In fact, it can do this in any standard, JDBC-compliant SQL database.

Servoy Developer

When you are designing a solution and making changes to forms and methods, Servoy is saving your work in various files in your Servoy Workspace, which is simply a folder in your operating system's file system, typically on your local hard drive (find out more about the Servoy workspace here) .

If you are a developer working alone, this Workspace is all you need to manage your applications under development, but you will be limited in terms of version control. If you want to periodically save revisions of your solution (such that you can return to an earlier version if necessary), you can submit each revision of your solution to a *servoy_repository* db to which you will connect from Servoy Developer.

If you are part of a team of developers you should use the Servoy Team Provider feature to merge your changes into a version control system such as SVN or CVS. That features is beyond the scope of this document.

When you are ready to deploy your solution, you export it from Servoy Developer and import it into Servoy Server. This import process places your solution in a db called *servoy_repository* to which Servoy Server is connected.

It is technically possible for Developer and Server to share the same *servoy_repository* db, making it possible for you to deploy your solution by submitting it straight from Developer into Server, however from an operations standpoint this is not recommended.

Servoy Server

When Servoy Server is running and a user opens up a solution in a Servoy Client session, Servoy Server reads the solution from the *servoy_repository* (using SQL) and pushes it down to the Client. When it's time to populate forms with data, Servoy Server reads the data from whatever database(s) the solution calls for and pushes the data down to the Client. The business data managed in one solution (and indeed in one form) can span multiple databases and multiple database vendors. The repository, however, is always contained entirely in a single database. Servoy Server is written in Java and runs on top of the Tomcat application server.

Servoy Client

The piece that users interact with directly is the Servoy Client. This can be either the Java client or the Web client. More on this in a moment.

The Java Client (sometimes called the Smart Client or the Rich Client) is a Java application that runs on top of a Java Virtual Machine on the end-user's computer, providing users with a "rich UI experience".

Under the Web Client, users access a servoy solution via a web browser. The Servoy Server renders the forms using HTML, JavaScript, AJAX and other standard browser protocols. Due to limitations of browser technology, there are some UI features that are available to the Smart Client that aren't available to the Web Client.

Beans & Plugins

In addition to these major components of the architecture, Java beans and Servoy Plugins can be used to extend Servoy's functionality. These need to be placed in the appropriate folders on the Servoy Server. Servoy Server takes care of downloading them to the client machines as necessary (including updates when new versions of the beans and plugins are installed) so that all plugin and bean administration takes places on the server, not on the client.

Batch Processes

Servoy Server also supports Batch Processes. These are Servoy solutions that don't have any user interface and which you want running continuously on the server. One use for batch processes is to schedule various jobs to fire at regular intervals using Servoy's built-in Scheduler functions.

by Adrian McGilly, Servoy Consultant & Trainer

Servoy Developer & Eclipse

by Adrian McGilly

What is Eclipse?

Servoy Developer is an Eclipse plugin. Eclipse is a standard software development platform which is included in your Servoy installation. Started by IBM and now an open source project supported by the Eclipse Foundation, Eclipse supports development in many programming languages and environments, not just Servoy. The Servoy plugin for Eclipse provides a full-featured Integrated Development Environment (IDE) for Servoy development. By working within Eclipse, Servoy makes itself more attractive to a huge number of enterprise developers who are already using Eclipse, thus promoting greater acceptance in the marketplace.

Eclipse is full of features that will make a Servoy developer's life easier, but there are a great many features in Eclipse which are of little or no use to a Servoy developer, so don't be intimidated by its depth and the huge number of commands and options in its various menus. Over time you'll figure out which ones are useful and which ones to ignore.

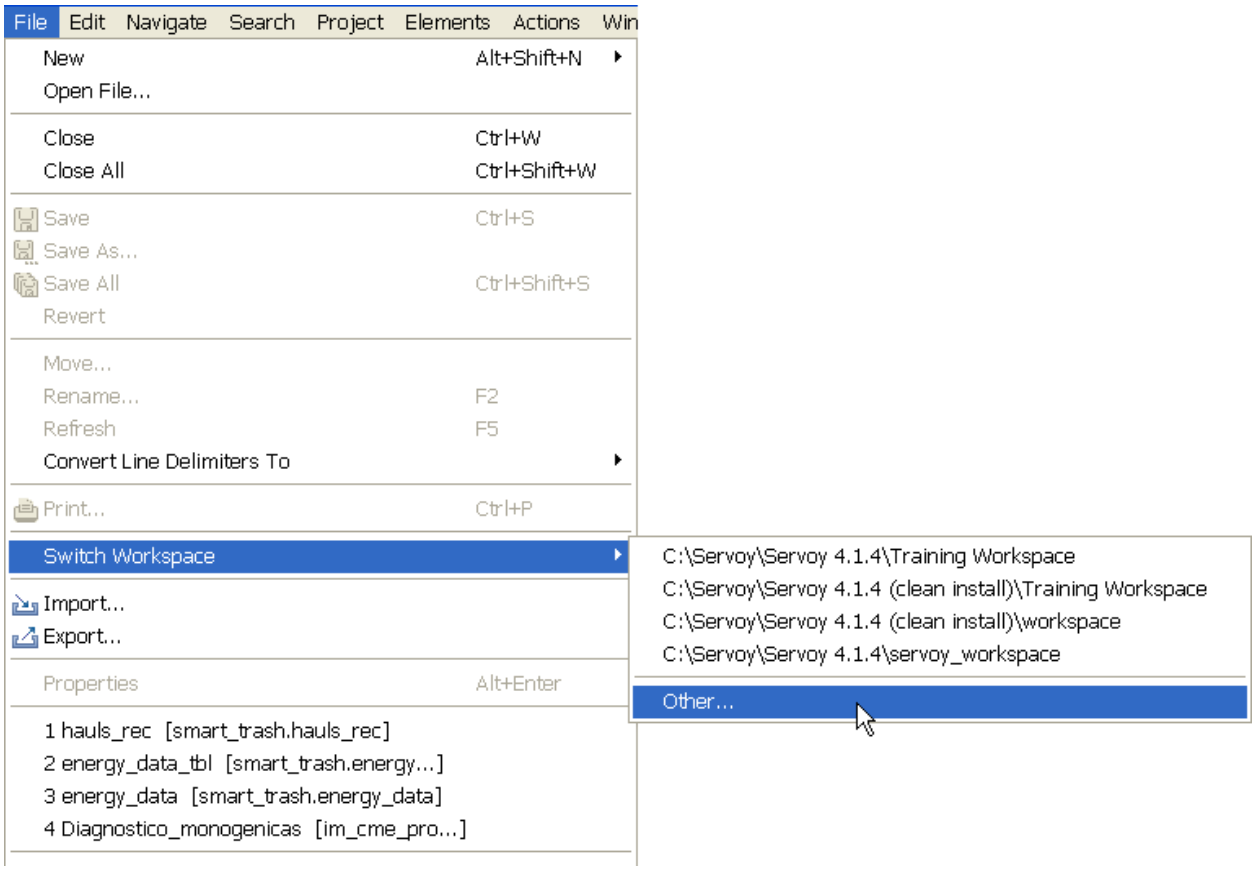
As you work with Servoy 4.x, sometimes you will be using Eclipse features, and sometimes you will be using Servoy features, and it won't always be apparent which is which. In this document, I won't attempt to make that distinction – I'll just say "Servoy does X" even if in fact it's Eclipse doing X for Servoy.

Here is a posting on the Servoy Forum describing useful features of Eclipse.

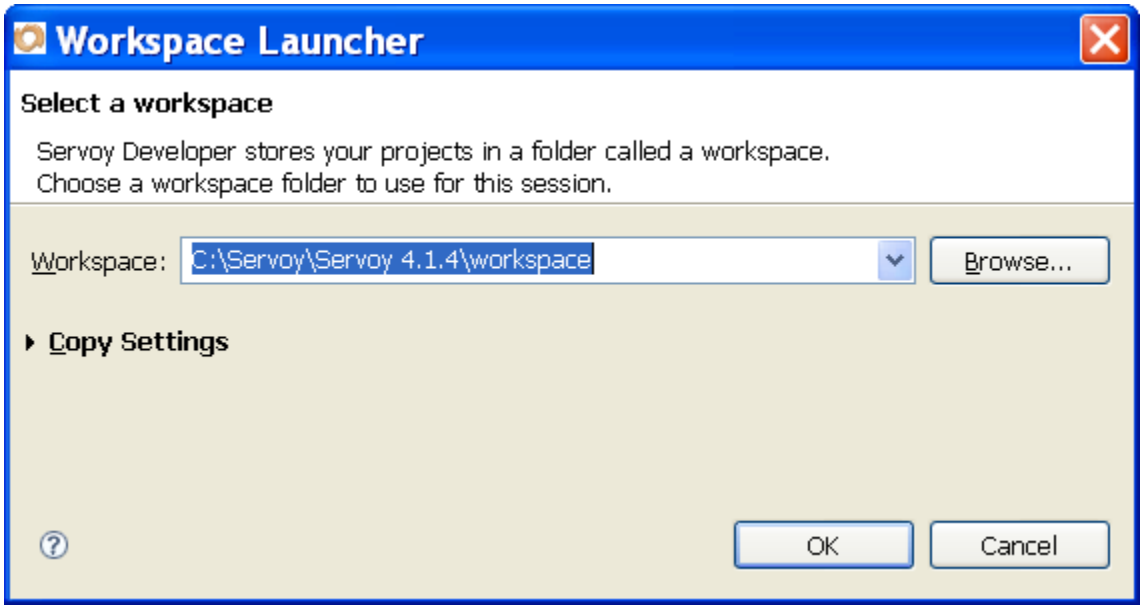
Workspace vs. Repository

Before you can begin working on a solution you have to get it into your Eclipse Workspace. The Workspace is a folder, usually on your local hard-drive, where Servoy stores your solution while you are modifying it. Certain preferences and settings for the Servoy IDE are also stored in the Workspace.

You can find out where your Workspace folder is by selecting File>>Switch Workspace>>Other…



This will display a dialog showing you what your current workspace folder is, and allowing you to change to a different workspace folder if you like:

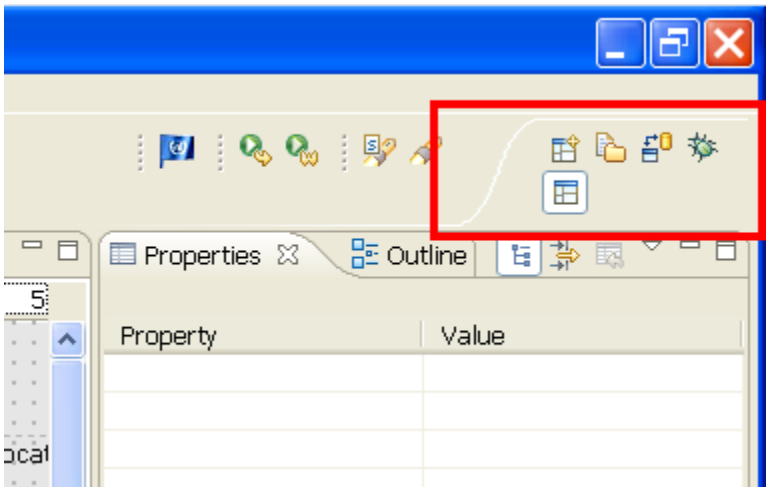


When you first start up Servoy, it will create a Workspace folder for you called servoy_workspace. Where it puts this folder varies by operating system. I personally like my Workspace folder to be inside my Servoy folder. This way I know that if I make a backup of my Servoy folder, I know I'm backing up everything, including any solutions I'm working on.

Eclipse Perspectives

In Eclipse, a "Perspective" is a set of panes that together support a certain type of work such as Form Design, Debugging or Synchronization. Servoy has already created some perspectives for you, which you can see if you select Window>>Open Perspective>>Other…

You can also switch among any perspectives you have visited recently using the Perspectives toolbar :



by Adrian McGilly, Servoy Consultant & Trainer

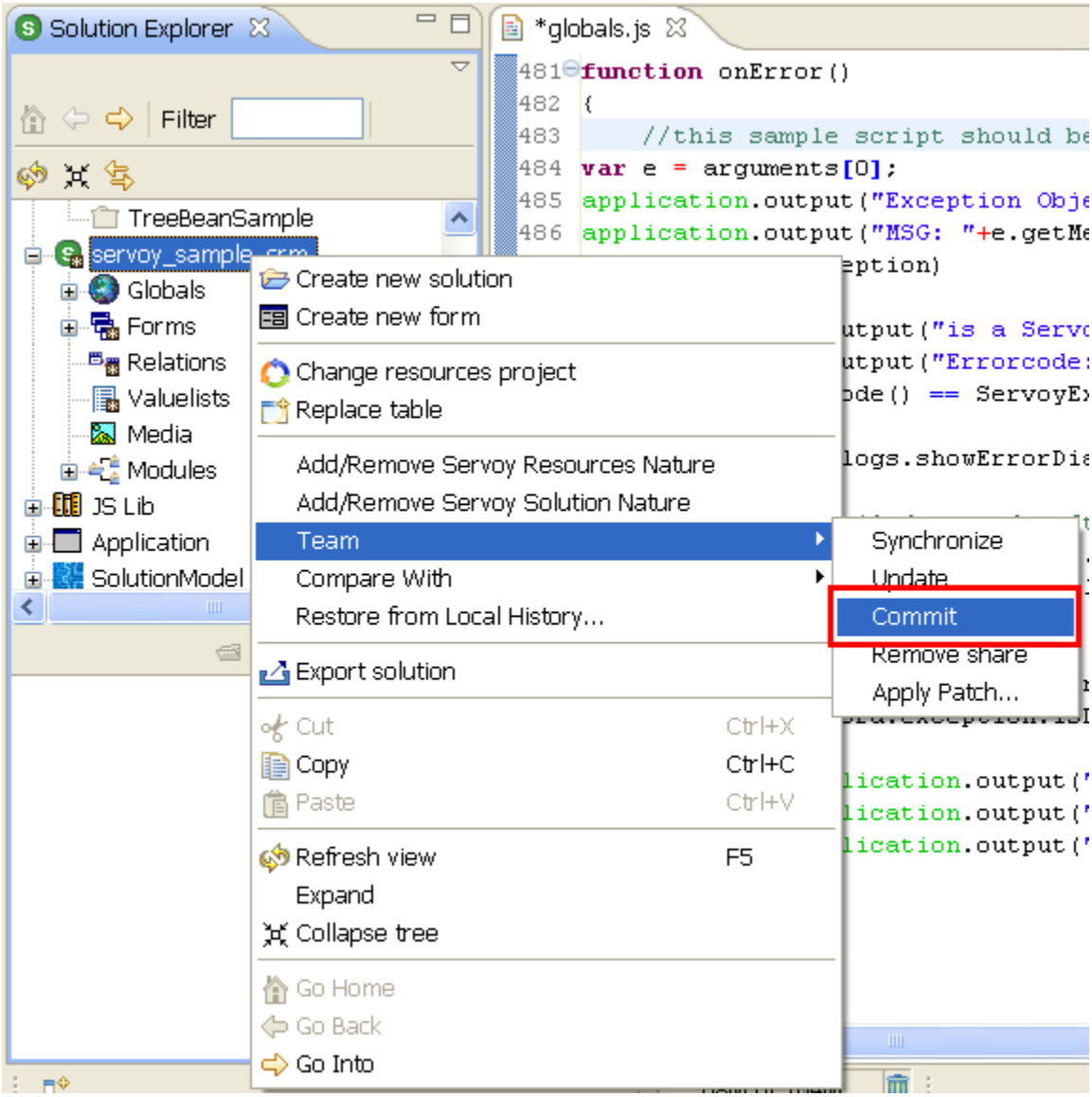
How do I save my changes to the Repository?

To save your solution in your local repository:

1. The solution needs to be the active solution. If it isn't already, activate it by right-clicking and selecting 'Activate solution'.
2. Right-click on the solution and select Team>>Commit. If the 'Commit' command isn't available then select 'Share Project…' and instruct Servoy to

share the project using the servoy_repository db. Now the Team>>Commit command should be available for you to select.

3. If you are lucky, your solution will get committed without requiring any further steps. If you are working as part of a team of developers sharing the same repository then there may be conflicts requiring resolution before your changes can be committed. To help you resolve these conflicts Eclipse has a 'Synchronization' perspective which you will be thrown into if conflicts are detected. I have found that even as a lone developer with sole access to my repository, Servoy often perceives conflicts between my new version (the one I'm submitting to the repository) and the latest version in the repository. Obviously this makes no sense seeing I'm the only one working on the solution and I can't be in conflict with myself. This is something the engineers at Servoy are aware of and have promised to fix in a future release. So if this happens to you, here are some workarounds to this problem.
    a. Go back to the Form Design perspective and try committing a second time. Sometimes the second time it works without finding any conflicts.
    b. If that doesn't work the from the Synchronization perspective, look in the list of conflicts for any bearing a red flag. Right click on each such item and select 'Mark as merged'. Now return to the Form Design perspective and try committing your solution again.



Here is an article on the Servoy Forum which describes the normal lifecycle for submitting code to the repository. A lot more could be written about this topic but it's beyond the scope of this document.

Connecting to SQL Anywhere Databases

by Adrian McGilly

About SQL Anywhere

SQL Anywhere is just one of many dbs that Servoy can connect to, but seeing it's the one that's bundled with Servoy, it's the one I'll assume you're using throughout this handbook.

SQL Anywhere has many names. It is sometimes called ASA for Adaptive Server Anywhere, and it is sold by a subsidiary of Sybase called iAnywhere and is sometimes called iAnywhere for that reason.

The license agreement Servoy has with Sybase allows you to deploy solutions using the Sybase SQL Anywhere database without having to pay any license fees to Sybase, provided no other software is connecting to the SQL Anywhere db. Given the depth, robustness and scalability of the SQL Anywhere db, this is a great value.

Before getting too far into Servoy, you will want to download a copy of the database management toolset for SQL Anywhere, called "SQL Anywhere Developer Edition" the main component of which is called Sybase Central. Sybase Central lets you manage your database schema, tweak column attributes like NOT NULL, INDEXED and UNIQUE, browse your raw data, perform adhoc SQL queries, create new databases, etc.

To read about SQL Anywhere, visit this link.

To download a free copy of SQL Anywhere, visit the Download Sybase Central page on the Servoy Website or go here: http://www.ianywhere.com/downloads/ and click on SQL Anywhere Developer Edition

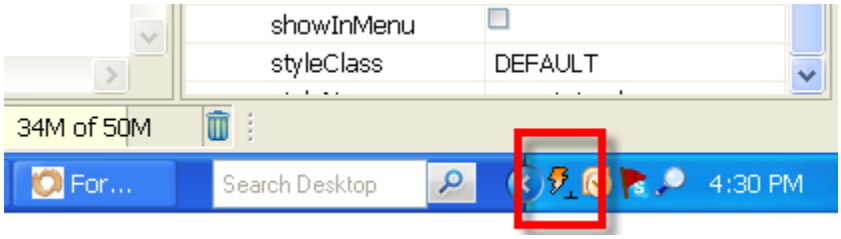
Connecting To SQL Anywhere Databases

There are a few details that need to be precisely in place for Servoy to successfully connect to SQL Anywhere databases. Generally, after installing Servoy you should

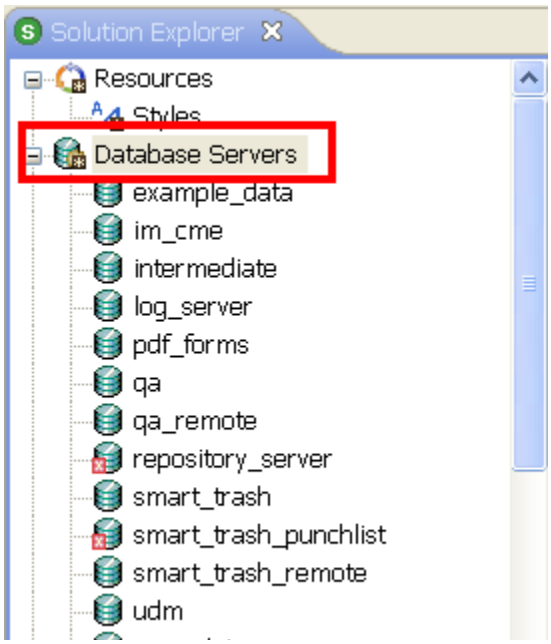
be able to connect without any problems, but there have been installers in the past where this wasn't the case, so here are some pointers.

When Servoy starts up, it starts the SQL Anywhere database engine, which is contained in the servoy/application_server/sybase_db folder. SQL Anywhere then looks in the file called sybase.config for a list of databases it expects to find and it tries to open each one in the list. If ANY of these don't open successfully, Sybase quits (however Servoy sill opens – it just won't be able to connect to any databases and therefore won't work very well).

If you're developing on a Windows machine you can tell if SQL Anywhere is running by whether its icon appears in the icon tray (see below):



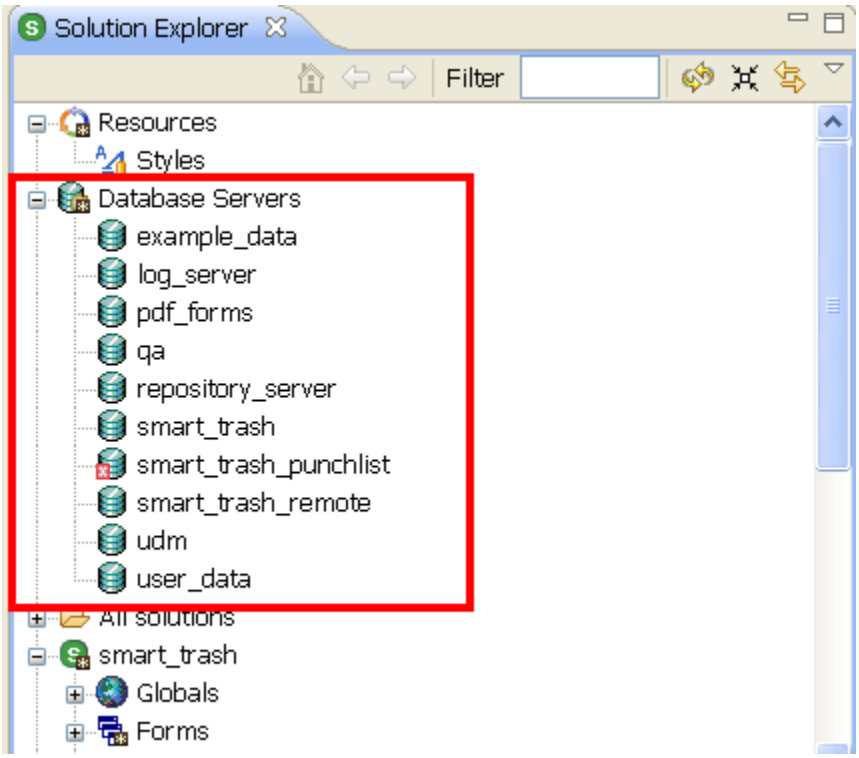
On a Mac look in the Activity Monitor for a process called dbsrvXX where XX is the version of Sybase you are running.



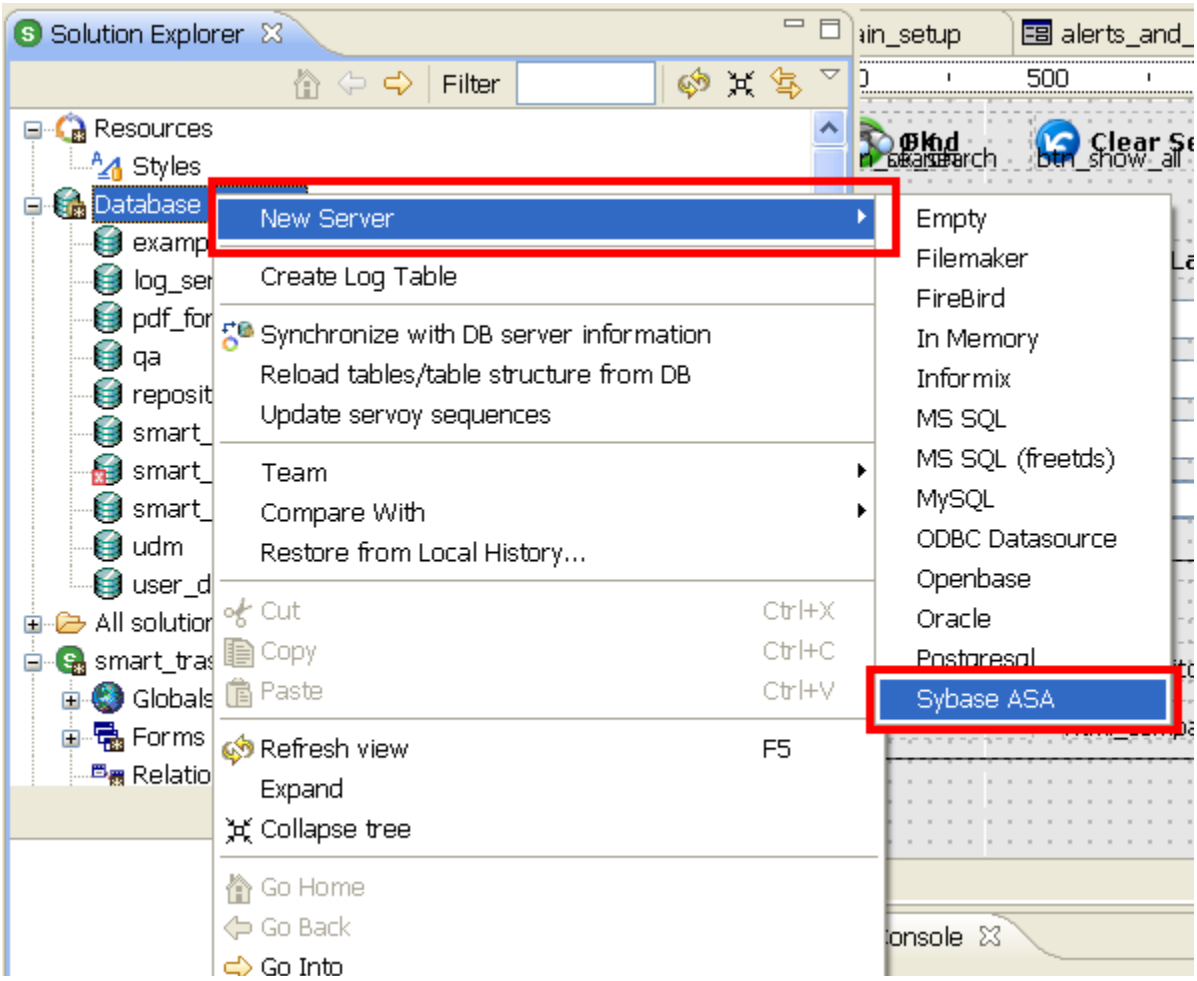
If you don't see any database servers listed under the Database Servers node of your Solution Explorer, then something went wrong with Sybase. To resolve this, quit Servoy, open the sybase_log.txt file in the sybase_db folder and scroll to the bottom. Usually you will see there a message telling you what's wrong, such as "Could not open/read file: database/mydatabase.db".

Note: it's OK if there are databases in the database folder OTHER than the ones specified in the sybase.config file. What's not OK is if there is a database specified in the sybase.config file that is NOT in the database folder.

Once Sybase has successfully opened a database, Servoy needs to be told what databases to connect to and how. These database connections are simply called "Database Servers" in the Servoy IDE, and they appear in the Solution Explorer under the Database Servers node (see below).



In order for Servoy to connect to a database it must have a server connection defined in this list. To create a new connection right-click on Database Servers and select New Server, then select the brand of db you are connecting to:

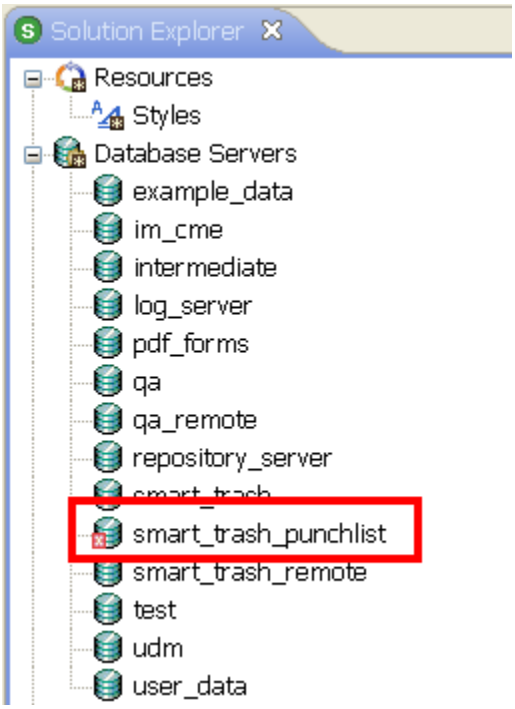


Now fill in the settings for the new db connection. Note that if your Sybase database is called crm.db, then you will put 'crm' (no quotes) as your database name (not 'crm.db'). Servoy will only let you save this new connection if it can successfully connect right then.

More details on how to create these connections are in the Servoy Developer User's Guide.

These connection settings are all stored in your servoy.properties file, which you will find in the application_server folder of your Servoy folder. It can happen that you have more connections defined than you have databases opened. In such cases, in the list of Servers under Database Servers, there will be a red X next to

any database connection which has failed to connect to its designated database (see example below).



Troubleshooting Connecting To SQL Anywhere:

Corrupt or out-of-sync .log file, corrupt database

Each database file should have a .log file by the same name (e.g. the log file for crm.db is crm.log). It can happen that the log file gets out of sync with the db file. In such cases, you can try deleting the log file and SQL Anywhere will create a new one the next time it successfully opens the database.

If that doesn't work, try forcing Sybase to start up the database without a .log file by using the '-f' option.

1. On Windows, the procedure is:
2. Shut down Servoy and any running Sybase servers.

3. Run the dbsrv10.exe application in the servoy\application_server\sybase_db folder

4. Specify the .db database you are trying to open, and put -f in the options field

5. Click OK.

On a Mac, the procedure is:

Open up a new window using /Applications/Terminal and enter these commands (assuming your Servoy directory is located in /Applications/Servoy):

```
cd /Applications/Servoy/application_server/database

export
DYLD_LIBRARY_PATH=/Applications/Servoy/application_server/sybase_db

export
PATH=$PATH:/Applications/Servoy/application_server/sybase_db

dbsrv10 -f servoy_repository.db
```

SQL Anywhere Version Mismatch

I have seen it happen that the error reported in the sybase_log.txt file is something like "capability 32 missing" or "capability 35 missing". These usually have to do with version mismatches between the db files and the SQL Anywhere db engine. This should never happen on a fresh install of Servoy, but it can happen later down the road if for instance you open a db in Sybase Central using a more recent version of the dbsrv10.exe database engine than the one that came with Servoy. If this happens, replace Sybase files in the Servoy install with those from the Sybase Central install and that should resolve the mismatch.

Firewalls

Firewalls can interfere with Servoy connecting to SQL Anywhere. If you can turn off your firewall at least while you're troubleshooting getting

connected that will help. If you need to use a firewall, look in your firewall settings for any rules blocking access to/from dbsrv10.exe and remove these rules.

Corrupt Database

If any of the dbs listed in the sybase.config file are corrupt that will cause problems. You can often decorrupt these by running the dbsrv10.exe app and using the –f parameter to "force" the database.

Need to restart SQL Anywhere and Servoy

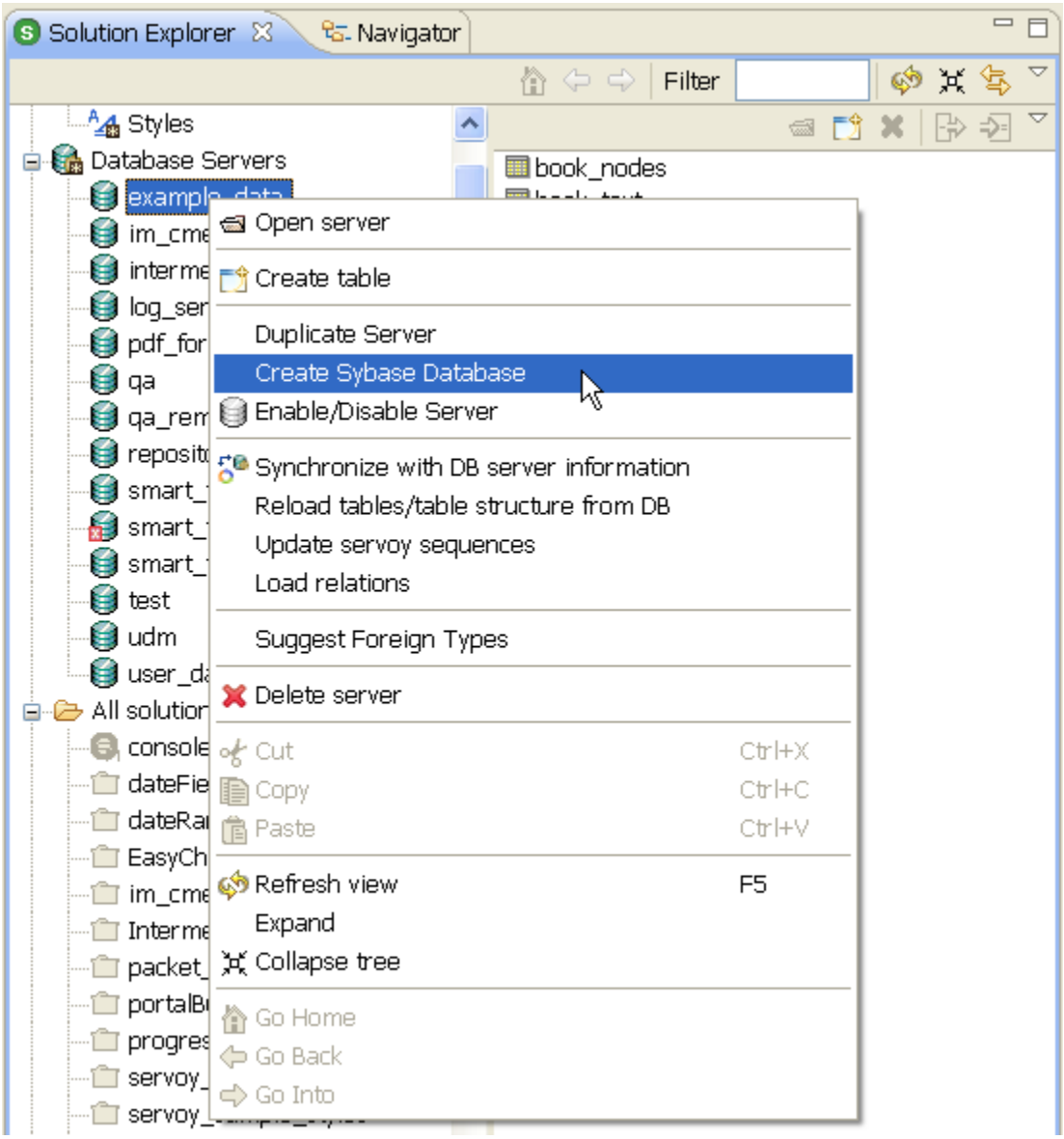
If you change settings in the sybase.config file, they will only take effect the next time SQL Anywhere starts up. If SQL Anywhere is already running and you start up Servoy, your changes to sybase.config will be ignored. So after making changes to this file, make sure you exit SQL Anywhere before you start up Servoy. To shut down SQL Anywhere in Windows, find its icon in the icon tray, right-click and select exit or shutdown (or on a Mac, select 'Quit' after selecting 'dbsrv10' using the Activity Monitor).

Similarly, if you think you've got everything set up right but Servoy is still refusing to connect, always try quitting out of both Servoy and SQL Anywhere and then restarting Servoy before giving up – Servoy uses a "lazy load" approach to starting up and this can mean that certain changes don't take effect until you restart.

Creating A New Database

Once you have Servoy connecting successfully to SQL Anywhere, you will want to create a new solution and start playing around. What database should you use for that?

You can use one of the existing sample databases that come with the initial Servoy installation (at this time they include example_data.db, user_data.db, crm.db and udm.db) or you may want to create a new database. To do this, right-click on one of the existing databases and select 'Create Sybase Database' and follow the instructions.

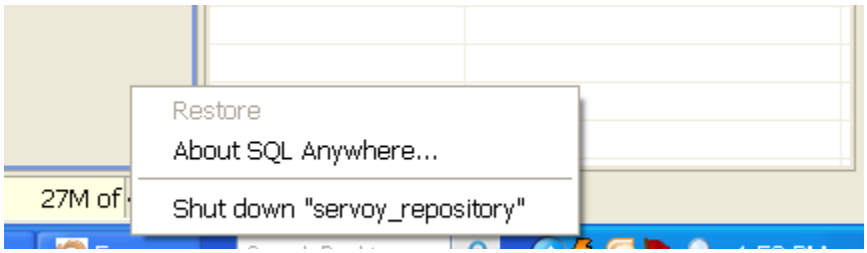


The default username and password for your new SQL Anywhere database are:

username=dba
password=sql.

Shutting Down Sybase Server

To shut down Sybase on Windows, right-click on the Sybase SQL Anywhere icon in the system tray and select Shut Down "servoy_repository":



On a Mac, locate the dbsrvXX process in the list of active processes in the Activity Monitor and hit the Quit button.

Be patient - it can sometimes take Sybase a while to quit as it cleans up its transaction log. DO NOT do a force quit of this process unless you are prepared to lose data.

How Can I See My Raw Data Outside of Servoy?

One way to see your raw data is to use Sybase Central. Once you connect to a db from within Sybase Central you can drill down to a particular table, select the 'data' tab and you'll be looking at raw rows and columns of data. Be aware that if you make changes to the data in Servoy, you will have to hit 'refresh current folder' in Sybase Central to see the changes reflected there.

There is also an interactive SQL Query editor built into Sybase Central that can be useful (though it's not hard to build such a tool in Servoy - there's one built into the svyCRM demo solution, called AdHoc Query under the Admin tab).

Eclipse also supports plug-ins for viewing SQL data right in Eclipse. Sybase and other vendors offers a number of Eclipse-based database design and administration tools which may be of interest to you.

by Adrian McGilly, Servoy Consultant & Trainer

Where is my database schema stored?

by Adrian McGilly

Tables And Columns

Servoy lets you do a small amount of database management from within Servoy itself: you can create and delete tables and columns. For any other changes to the schema (e.g. modifying attributes of columns, renaming or deleting columns, renaming tables, etc.) you need to use Sybase Central.

When a Servoy solution is opened, it loads all db schema info from the db. If you go into Sybase Central and modify the db schema (e.g. add a column, modify a column's attributes, add a table) you will see your changes in Servoy's dataprovider window the next time you start up Servoy. (It's best to shut down SQL Anywhere and restart Servoy after making such changes, although in some cases it's not necessary.)

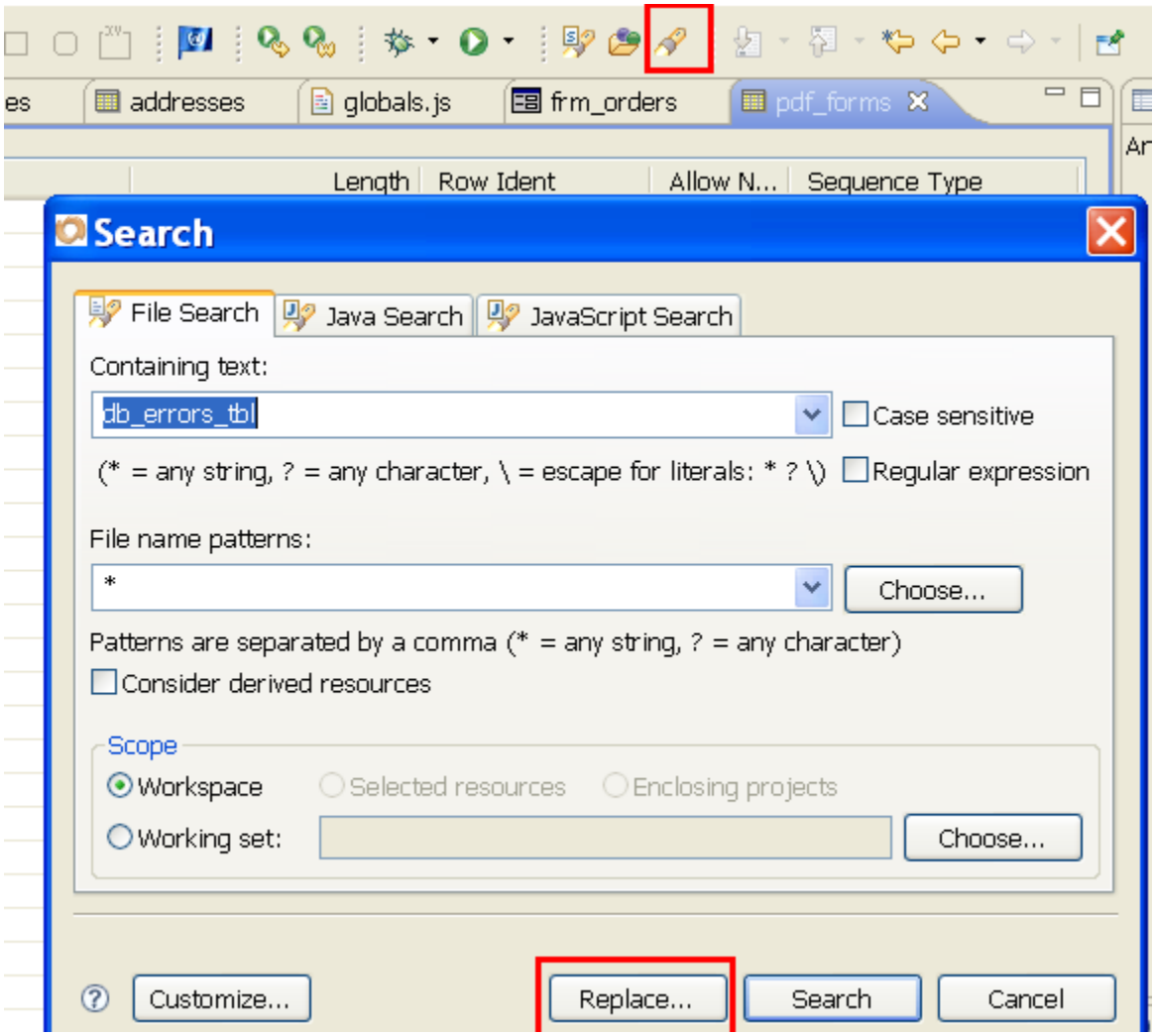
You need to be mindful when you do this so as not to break any code that refers to tables and columns.

Some examples:

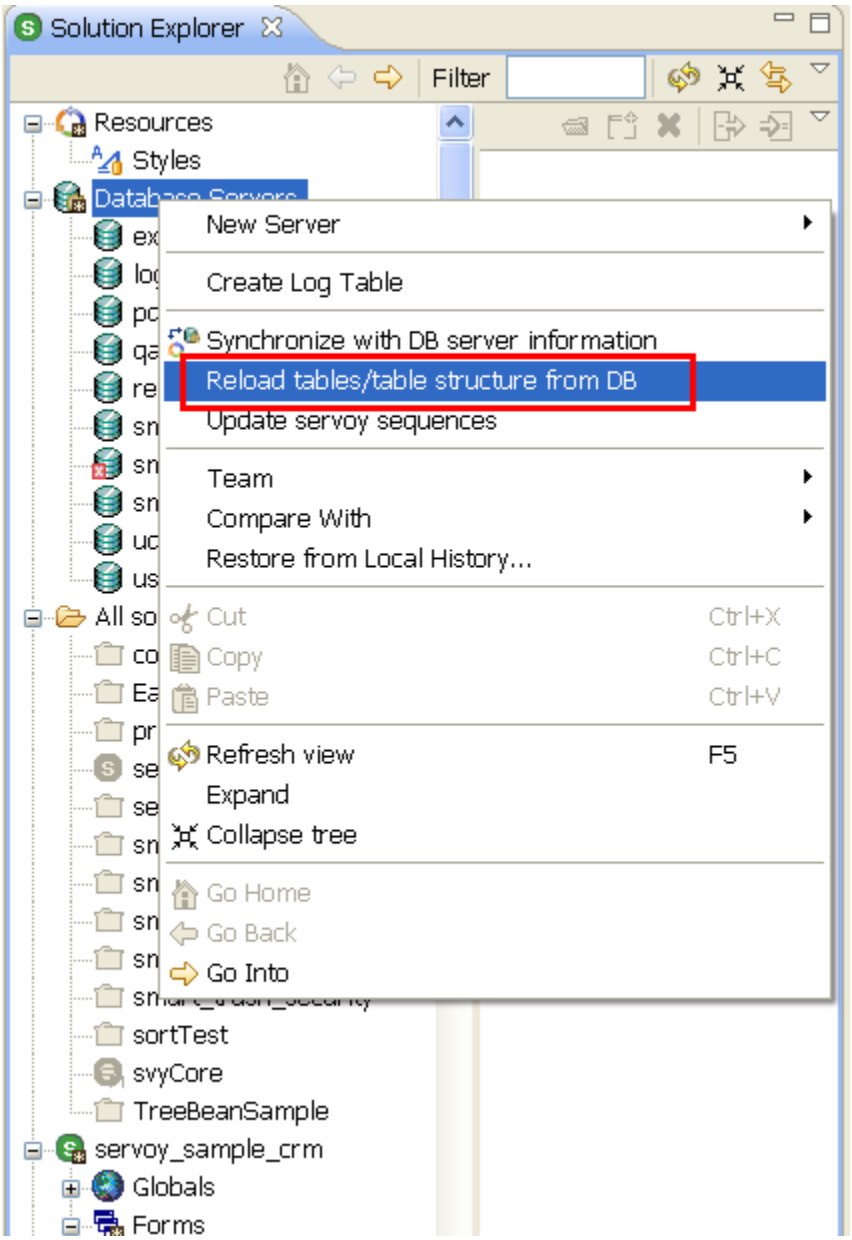
Change To Backend DB	**Effect on Servoy**
Rename a column	The new name will appear in the table editor in Servoy the next time you start up Servoy but any fields on forms that referred to that column by the old name will display as empty – they remember the old name – and methods that refer to the old name will cause errors, so be careful.
Delete a column	The change will be reflected in the table editor in Servoy the next time you start up Servoy; any fields that referred to the deleted column will display as empty;

	methods that refer to the deleted column will cause an error.
Modifying a column's attributes	The change will be reflected in the table editor in Servoy the next time you start up Servoy; adverse affects to fields displaying that column or methods referring to that column are possible, depending on the kind of change you made.
Detele or rename a table	If you delete or rename a table and that table is referred to in your solution you will get an error the next time you open your solution, either at startup or the first time the solution refers to the table.

Note that there is a global find & replace command in the Servoy Editor that will let you make global changes to your JavaScript methods:



Also, if you've made changes to the db schema outside of Servoy while Servoy Developer is running, you can load those schema changes into Servoy without restarting Servoy by right-clicking on the Database Servers node in the Solution Explorer and selecting 'Reload tables/table structure from DB:



Sequences

You can define sequences from within Servoy. One type of sequence, called a "Servoy Sequence" enlists Servoy Server to manage the generation of sequence numbers (as opposed to a "database sequence" where you are telling the database server to generate the sequence numbers). In the case of Servoy Sequences, Servoy stores information ABOUT each sequence (such as the "increment by" amount and

the next number in the sequence) in the repository, but the sequence numbers themselves remain in the database.

by Adrian McGilly, Servoy Consultant & Trainer

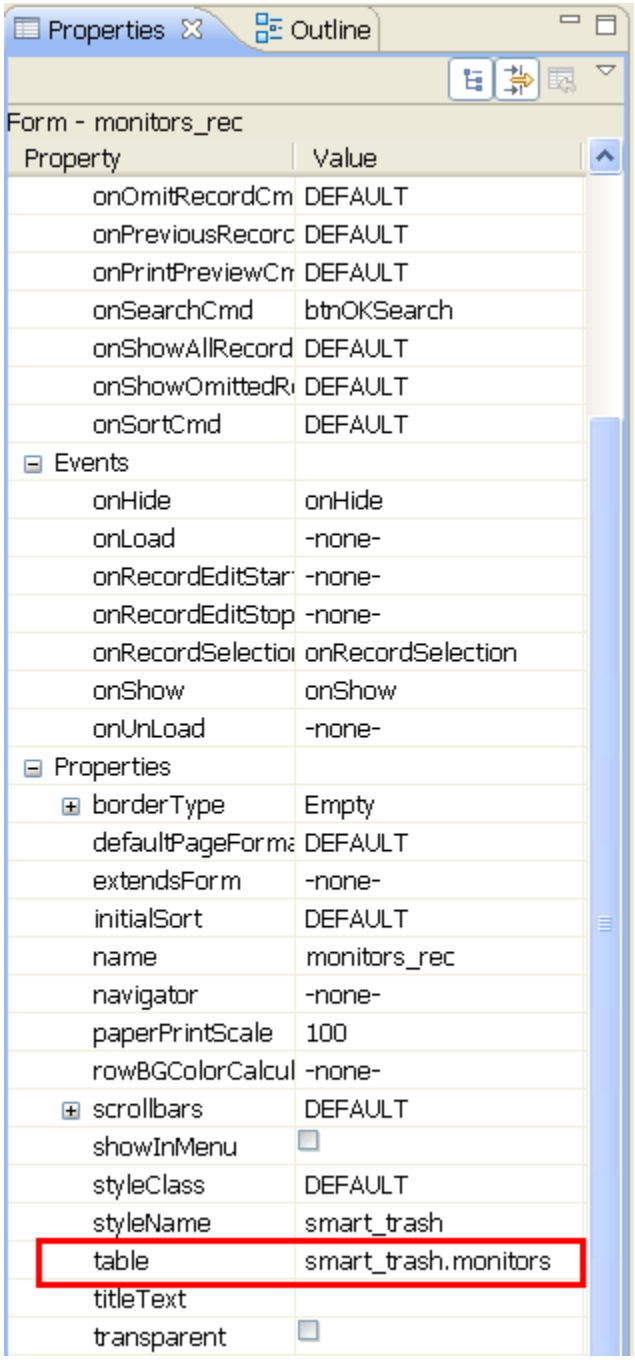
Foundsets

by Adrian McGilly

What Is A Foundset?

Each time a Servoy solution requests data from the database, a Foundset is created containing the results of the query. We'll look at the details of how and where this is done a bit later, but for now just think of a Foundset as a collection of rows that have been retrieved from the db, and which the solution can now 'play with'. Once rows of data are contained in a foundset, Servoy offers many ways to display, sort, edit, delete, print and generally process that data. Foundsets are you friends.

When you create a form in Servoy you tell it what db table the form is based on. This setting is stored in the form's "table" property (shown below) and can be seen in the properties panel when viewing the properties for that form.



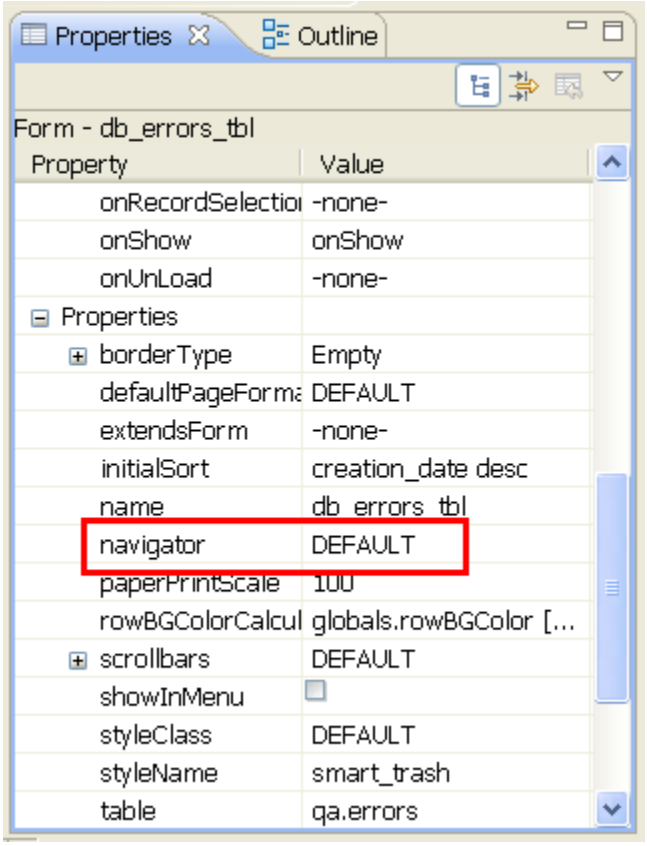
This doesn't mean you are limited to working with that table's columns on the form, it just serves as a starting point. (As you will see, you will use Relations and Tabpanels to pull in data from other tables, but more on that later.) By default, all forms that are based on the same database table will share the same foundset. As a result, two different forms that are based on the same table will by default display

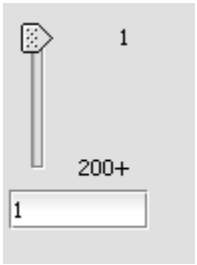
the same records when opened, and furthermore the exact same record will be "selected" or "current" in both forms.

In case that sounds a bit limiting, you'll be glad to know there's a property on each form called *UseSeparateFoundset* which, if set to true, lets each form have its *own* foundset.

A good way to get a feel for how foundsets behave is to experiment with a table that has at least 500 rows of data in it. Fortunately, the example_data sample database that comes with the installer contains several tables with 500+ rows in them (e.g. the orders and order_details tables).

Once you have the 500+ row table and a form based on that table, start fooling around with searches, inserts, saving data, list views, record views, table views, etc. Be sure the 'navigator' property of the form is set to 'DEFAULT' (as illustrated below) so that you get the default slider control (also illustrated below) for navigating records in your foundset. The numbers on this slider are quite informative.





Let's look at an example. Let's say you have a Companies table with 500 records in it. You create a customerForm form based on that table and you open the form up in runtime mode.

When you open a form in Servoy, by default it will create a foundset comprising all 500 rows in the table sorted by primary key (PK), but at first it will only retrieve the first 200 rows. Let's say you perform a search that comprises 450 rows. Your *foundset* now contains 450 rows, but again Servoy will start by retrieving just the first 200 rows in the foundset. Let's call those 200 rows the *partial foundset.*

The moment you select the 200th row of your *partial foundset*, Servoy automatically goes out and retrieves the next 200 records, so you now have a *partial foundset* of 400 out of a total foundset of 450 rows from a table that has 500 rows.

The moment you select the 400th record in your *partial foundset*, Servoy will go out and retrieve the remaining 50 records, making your *partial foundset* equal to your *foundset.*

How Do Foundsets Work?

What's really going on behind the scenes here?

To answer that question we need to look again at the architecture of Servoy. Remember that in a production situation, there are three tiers working in concert: the database server, the Servoy Server and the Servoy Client. (In development mode, there is no Application Server – the Servoy development environment is in fact a local instance of Servoy Server.)

Each time you create a foundset in Servoy, you are creating an object in Servoy Server's memory that keeps track of a whole bunch of things. For starters, it retrieves a list of primary keys (PKs) for all rows that match the search, and it holds that list in memory. It also keeps track of:

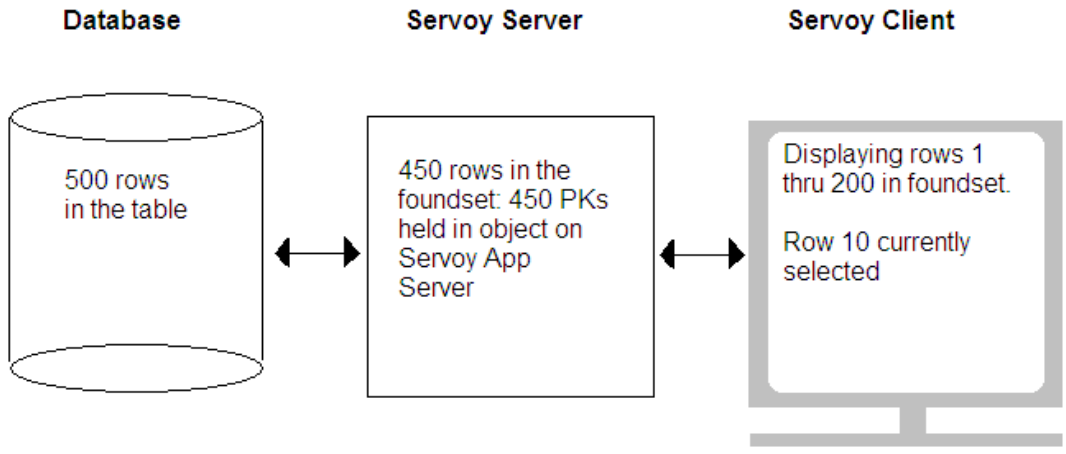
- the "query" that created the foundset,
- the total number of rows in the foundset,

- what rows are currently *displayed*, and on what form(s)
- which row of the foundset is currently *selected*.
- which rows you've modified or deleted since the foundset was created
- what other running Servoy clients are looking at those rows (i.e. in a multi-user context)
- etc.

So when you perform a search that returns 450 rows, Servoy is building a list ON THE SERVER (NOT in client memory) of the 450 PKs in the foundset. It feeds the first 200 PKs to the client, as well as all the data columns for just those rows that need to be displayed on the form.

As you navigate through the foundset, the client knows what records need to be displayed and it requests their data columns from the Servoy Server on an as-needed basis, using the PKs provided by the server. When you hit a multiple of 200 in the foundset, the server sends the client another block of 200 PKs and the whole process continues.

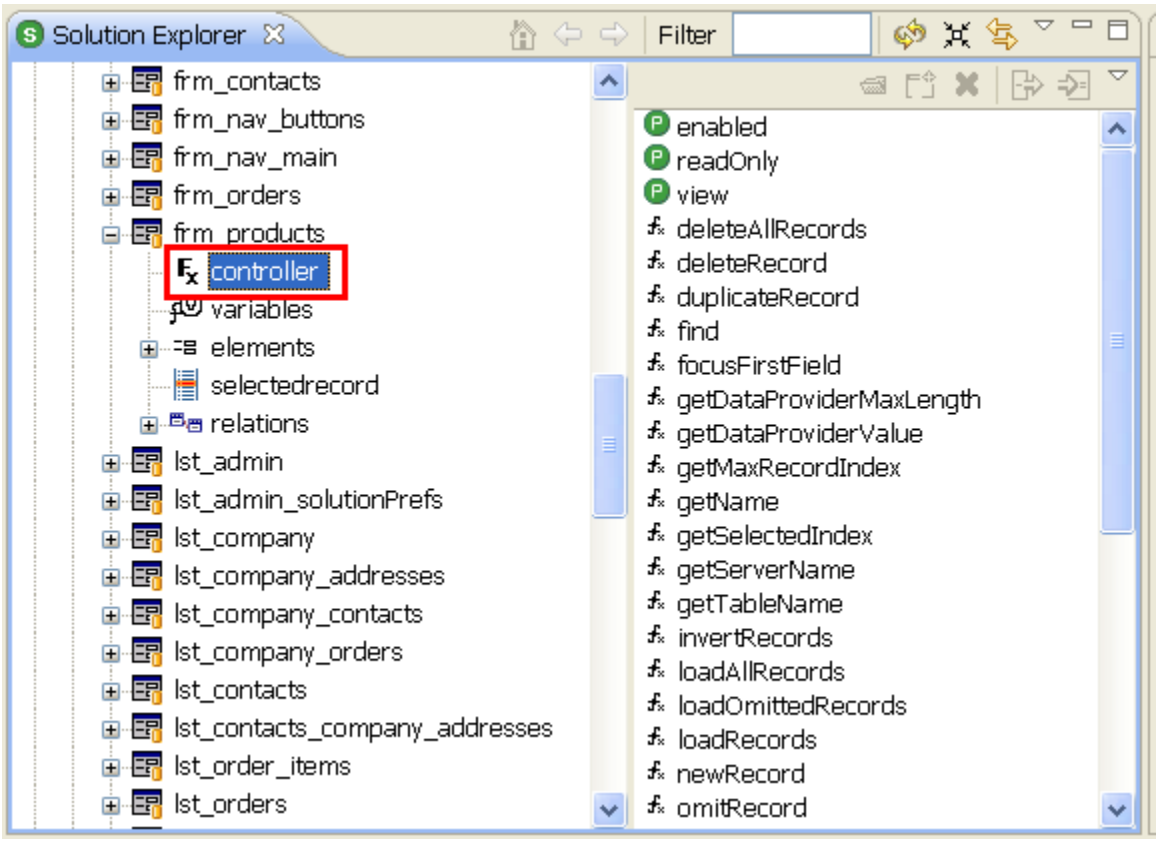
Returning to our example, let's say that Servoy has retrieved the first 200 records of your 450-record foundset and you have selected the 10th record. Here's an illustration of what's going on:



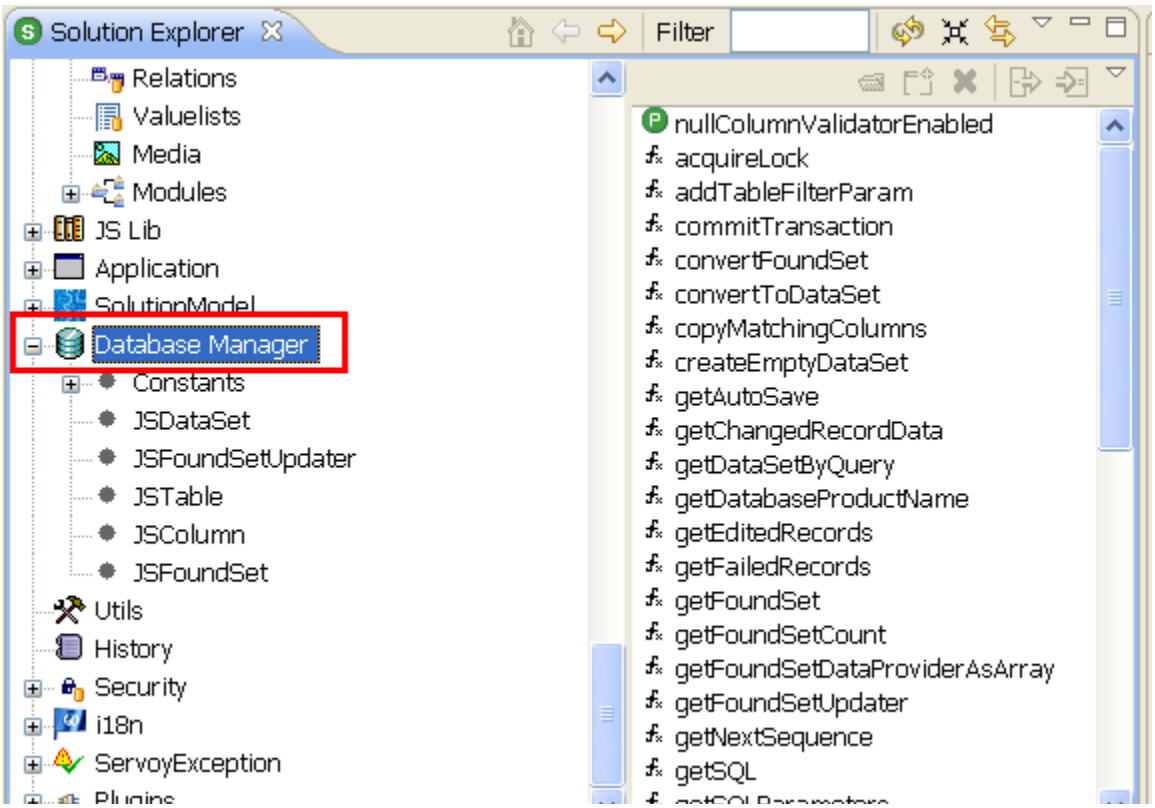
How To Programmatically Navigate A Form's Foundset

Now we know a bit about forms and their foundsets, how the foundsets are created and how they work. What tools are available to the programmer to navigate and process a form's foundsets?

There are many functions available to help you navigate a form's foundset. Most are under the form's controller node in the Servoy Editor (see below):



but a couple are in the databaseManager class, shown below:



For now, let's focus on a few key functions that give you most of what you need will need to navigate a form's foundset:

Function	**Description**
databasemanager. getFoundSetCount(foundset)	returns the total number of records in your foundset (not just how many have been returned so far to the client, but how many there are in all). In our example, that's 450.

Function	**Description**
controller.getMaxRecordIndex()	returns the number of records in your *partial* foundset (i.e. the number of rows for which the client has been fed the PKs so far). In our example that started out being 200, then jumped to 400 and then ended at 450. You can get the same result with foundset.getSize().
databasemanager. getTableCount(foundset)	returns the number of records in the table that the foundset is based on. In our example that's 500
controller.getSeletedIndex()	returns the number of the currently selected row in the foundset, which in our example is 10
controller.setSelectedIndex(n)	selects the nth record in the foundset (explained further below)

Perhaps the most important of these commands is the last one, controller.setSelectedIndex(n), which selects the nth record in the foundset. (Foundset rows are indexed starting at 1, so setSelectedIndex(5) selects the 5th row of the foundset).

What does it mean to 'select the 5th row of the foundset'? It means two things:

- Visually, that row will become displayed and selected on the form displaying the foundset.

- All values from all columns of that record will be loaded into the Servoy dataproviders for that table, making those data values available to your methods for reading and writing.
- If the row you've selected is a multiple of 200, it will trigger the retrieval of the next 200 records

Going back to our example, let's say you've just performed the search that retrieved the first 200 records in a 450-record foundset. If you did

```
controller.setSelectedIndex(199)
```

you would be selecting the 199th record in the foundset, loading its values into the dataproviders. If you did

```
controller.setSelectedIndex(200)
```

you would not only be selecting the 200th record but you would also trigger the retrieval of the next 200 records into the *partial foundset*. Note however that if you did

```
controller.setSelectedIndex(250)
```

when you only have the first 200 records, nothing would happen. You can't select beyond the end of the currently loaded foundset, and you must select the precise multiple of 200 to trigger the retrieval of the next 200 rows.

Looping Through A Foundset Containing 200+ Records

A question that often comes up is, how do can I cycle through ALL the records of a 200+ foundset if Servoy only retrieves 200 records at a time?

The easiest approach is to make use of databasemanager.getFoundSetCount() which returns the total number of records in the foundset. You can capture that value in a local var and then loop from 1 to that var. Here's the code:

```
//declares a local var and set it to the total number of
//records in the foundset
var max = databasemanager.getFoundSetCount(foundset)

for (var i = 1; i <= max; i++ )
{
     controller.setSelectedIndex(i);
```

```
	// do whatever processing you need to do here
}
```

Notice that we skipped the creating of the local var max and just done this:

```
for (var i = 1; i <= databasemanager.getFoundSetCount(foundset);
i++ )
```

But given Servoy's warnings (discussed above) about the potential expense of the getFoundSetCount() function it's best to avoid calling it at every iteration of a loop like that.

Here's another way to loop through all the records in a foundset:

```
var i = 1  //declares a local var and initializes it to 1

while ( i <= foundset.getSize() )
{
	//select row i and increments the value of i by 1
	controller.setSelectedIndex(i++);

	// do whatever processing you need to do here
}
```

or you could do this:

```
for (var i = 1; i <= foundset.getSize(); i++ )
{
	controller.setSelectedIndex(i);

	// do whatever processing you need to do here
}
```

In both of these cases, the moment the variable i hits a multiple of 200, the setSelectedIndex(i) call will trigger the retrieval of another batch of rows, and this will affect the value returned by getSize() in such a way that the loop will continue running until ALL records in the foundset have been selected.

How Adds, Edits & Deletes Affect Foundsets

When you add a new record using

```
controller.newRecord()
```

the record gets added to the current foundset immediately, but is not saved to the db until a save is performed. You can read more about saving data here.

Similarly, edits made to records are saved immediately to the foundset, but aren't saved to the db until a save is performed.

Deletes occur simultaneously in the foundset and the db.

By default, Servoy saves a record's changes to the db as soon as you leave the record, but you can modify this behavior in such a way that saving to the db is deferred until you explicitly request it. This means you might make multiple changes to multiple rows, (and possibly in multiple foundsets!) before any of these changes get saved to the db. The foundset in this case acts as a sort of staging area for making multiple changes to records before they are saved to the db.

Note: this behavior is not to be confused with db transactions, which give a separate level of control and are discussed here.

Caching & Data Broadcasting

Servoy Server does some very clever caching in order to minimize the number of times it has to communicate with the clients and the database, but it does this without compromising the freshness of the data. In fact, Servoy Server makes sure that any changes you make (and save) to any rows in your foundset are instantly reflected as necessary on any other forms in your solution and even *on the forms of other Servoy clients currently running the same solution!*

For example, if you change a unit price from $1.00 to $2.00, and that affects some calculated total appearing on a sub-form in a tabpanel, Servoy takes care of refreshing the subform automatically. Furthermore, if another user running the same solution is viewing a form that is affected by that change, Servoy Server will refresh their form automatically. Pretty amazing. (And as of version 3.5, data broadcasting works on the Web Client just as it does in the Java Client.)

by Adrian McGilly, Servoy Consultant & Trainer

JSFoundset Class

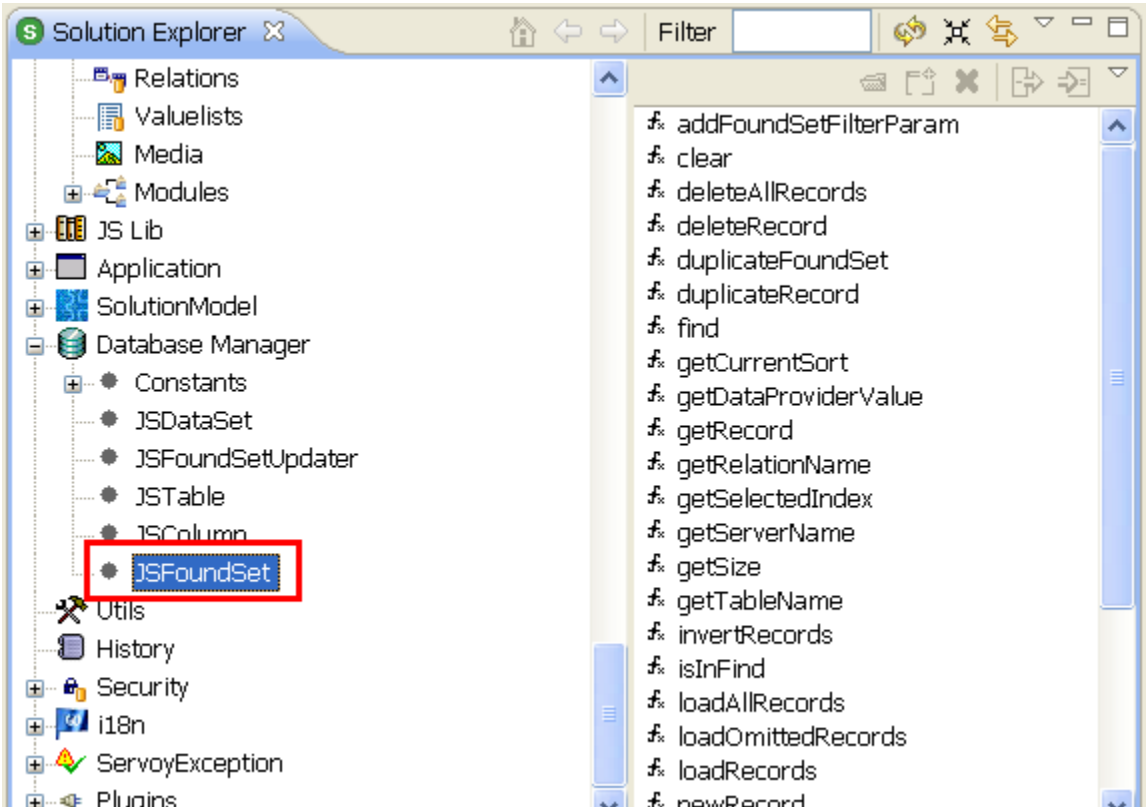
Accessing & Managing Data Without Relying on a Form

Thus far when referring to foundsets, I've been assuming the foundset belongs to a form, and that it can therefore be navigated using the functions under the form's controller. But what if you want to work with data without involving a form? For instance, what if the method you're writing for a button on your customers_rec form needs to perform a lookup in the orders table? Do you *have* to use a form based on the orders table to perform the lookup? Thankfully, you don't. That's what the JSFoundset class is for.

Under the databaseManager node you will find a function called getFoundSet(). This function creates a new foundset based on whatever server and table you specify, and lets you manage that foundset without ever involving a form. Here's how it works.

```
var myFoundSet = databaseManager('example_data','orders')
```

That line of code creates a new foundset called myFoundSet which contains all the rows in the orders table (though, just like when you open a form for the first time, the foundset starts off with just the first 200 rows of the table). You can now manipulate this foundset exactly as you would a form's foundset, except that instead of using the form's controller functions, you will use the JSFoundset functions found under the JSFoundset node (see below):



So for example, to select the 10th row in the foundset, you would say:

```
myFoundSet.setSelectedIndex(10)
```

(Note that you won't be able to get the Servoy editor to type all that for you the way you can with the controller functions – you'll have to type the word 'myFoundSet' and then double click on 'setSelectedIndex' under the JSFoundSet node.)

To reduce the foundset to the set of orders shipping to Anchorage, Alaska having freight costs over $50 you would say:

```
myFoundSet.find()
myFoundSet.shipcity = 'Anchorage'
myFoundSet.shipstate = 'AL'
myFoundSet.shipfreight = '>50'
myFoundSet.search()
```

To add a new record to the orders table, one that ships to San Jose, CA and has a freight cost of $25, you would do this:

```
myFoundSet.newRecord()
myFoundSet.shipcity = 'San Jose'
myFoundSet.shipstate = 'CA'
myFoundSet.shipfreight = 25
databaseManager.saveData()
```

and so on.

And although we haven't yet covered Servoy Relations, note that you can also use these foundset objects in conjunction with relations. So for example if you have previously created a relation called orders_to_employees and now you want to know the name of the employee who placed the currently-selected order in myFoundSet, the following expression would return the answer:

```
myFoundSet.orders_to_employees.employeename
```

Foundsets created this way do not persist the way form-based foundsets do. In our example, myFoundSet is a local variable that will disappear once the method in which it was declared terminates. So this technique is useful mainly for doing short-lived 'behind the scenes' work on the database that does not require the foundset to persist.

Using JSFoundSet Functions On A Form's Foundset

If you look closely at the set of functions under JSFoundSet and compare it with any form's controller functions, you'll see that there's a lot of overlap. It turns out that there are several ways of doing the same thing to a form's foundset.

For example, if a method on the customers_rec form needs to select the 10th record in the form's foundset, it could do this:

```
controller.setSelectedIndex(10)
```

Or this:

```
foundset.setSelectedIndex(10)
```

Both of the commands above do the same thing, but in the first case you are using the *controller function* whereas in the second case you are using the *JSFoundset* function. Because this is a *form method*, the foundset that is referred to in the second example is the foundset of the form the method belongs to.

If you were writing a global method that had to do this same task, you could say:

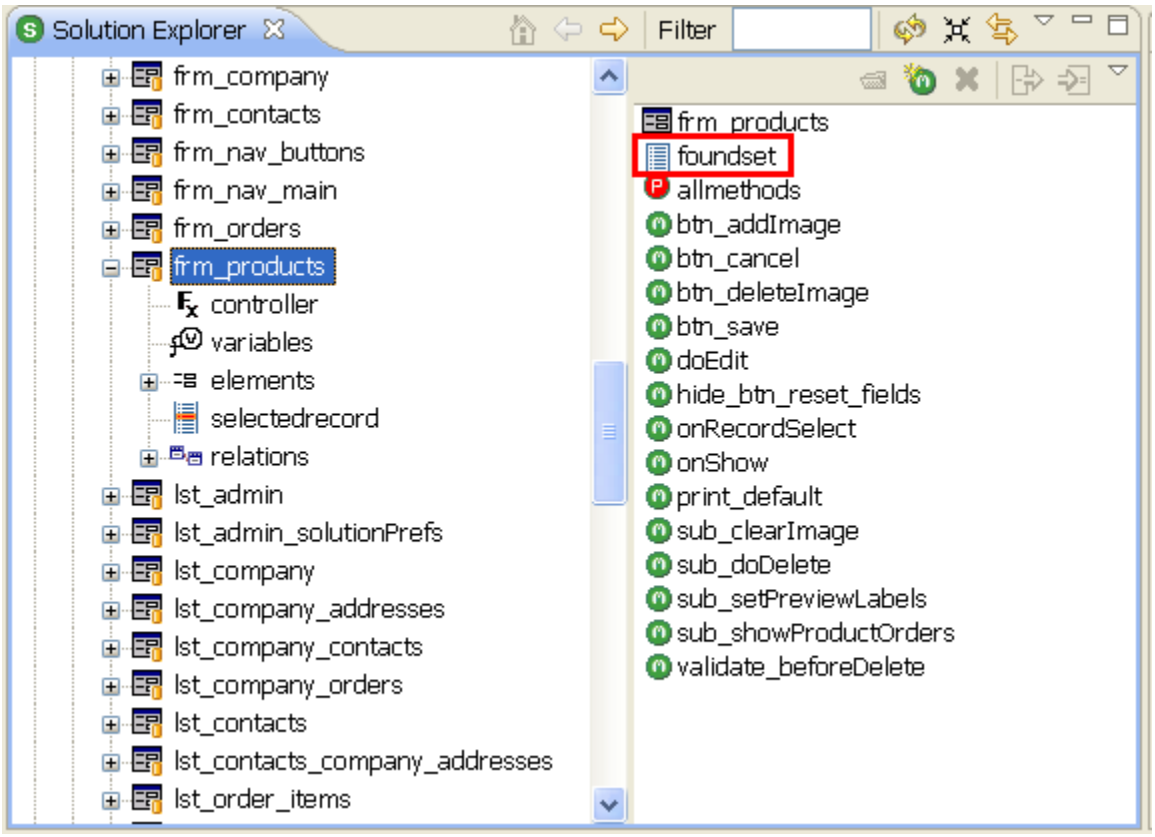
```
forms.customers_rec.controller.setSelectedIndex(10)
```

or you could say:

```
forms.customers_rec.foundset.setSelectedIndex(10)
```

and both would be equivalent.

A form's foundset object is represented in the object tree as seen below. Anytime you want to refer to a form's foundset, you can just double-click on this foundset object and Servoy will pop the correct reference into your method:



When should you use the 'foundset' object rather than the controller object to manipulate a form's foundset? It's up to you, but know that there are handful of functions in JSFoundSet which *aren't* available to a form's controller, such as

addFoundsetFilterParam(), getRecord(), selectRecord(), etc. So once you start needing those, you'll have to use the foundset object rather than the controller.

selectRecord(): Selecting A Record By Its PK instead of Its Index

One of the most useful functions in JSFoundset (which for some reason is NOT available under the controller object) is the selectRecord() function. There are going to be times where you want to select a certain record in a foundset, you know the PK value of the record, but you don't know what row it occupies in the foundset, so you can't get to it using setSelectedIndex().

You could of course do a 'find/search' on the PK value, but that would alter your foundset, reducing it to just that record.

But if you say

```
foundset.selectRecord('BALK')
```

that will select the record whose primary key value is 'BALK' (assuming such a record exists in the foundset) without making any other changes to the foundset. selectRecord() returns true if it finds the specified PK, and false if not.

The Servoy documentation for selectRecord is a bit confusing. It reads as follows:

```
//Select the record based on pk data
foundset.selectRecord(pkid1,pkid2...pkidn);//pks must be
alphabetically set!
```

The only situations where you would provide pkid2, pkid3, etc. is if the table the foundset is based on has a compound primary key, i.e. a PK that is based on several columns, and in that case, you are being told here to provide the values for each component of the compound primary key, in an order that matches the alphabetical ordering of the column-names. So if for example the order_details table has a compound primary key based on orderid and productid, to select the order_details record with orderid 100 and productid 999 you would say:

```
foundset.selectRecord(100,999)
```

getRecord(): Creating A Pointer To A Record in a Foundset

Another powerful function that is only available in the JSFoundset object is getRecord(). This functions returns a pointer to a specific record in a foundset, and you can save the pointer in a variable and know that it will always point to that same record, even if your code or your user selects a different record in the foundset. So for example, if you say

```
var myRecord = foundset.getRecord(10)
```

then the record in row 10 of that foundset is now pointed to by myRecord, and you can use that pointer to manipulate that record. For example if this is a record from the orders, you could set the freight to $100 like this:

```
myRecord.freight = 100
```

It may not be immediately obvious why you would use this function. Here are a couple situations where it is useful:

There may be times where you need to work simultaneously with different records in the same foundset. The ability to name a pointer to a specific record can make your code much more manageable and readable. If you do this:

```
var myRecordA = foundset.getRecord(10)
var myRecordB = foundset.getRecord(15)
```

Now you can refer independently to the 10th and 15th records of the foundset without having to mess with the setSelectedIndex() functions. For example you could copy the freight value from the 10th record to the 15th record like this:

```
myRecordB.freight = myRecordA.freight
```

Or let's say you want to modify several columns in a record which you are reaching via multiple nested relations (e.g. foundset.customers_to_orders.orders_to_order_detail). Your code for modifying the record might then look like this:

```
foundset.customers_to_orders.orders_to_order_detail.setSelectedI
ndex(1)
// now the current record in this related foundset
// is the first record in the foundset
foundset.customers_to_orders.orders_to_order_detail.order_date =
```

```
new Date()
foundset.customers_to_orders.orders_to_order_detail.unit_price =
100
foundset.customers_to_orders.orders_to_order_detail.freight = 20
etc.
```

This code is very verbose and thus harder to understand and maintain. It would more elegant and efficient to do this:

```
myRecord =
foundset.customers_to_orders.orders_to_order_detail.getRecord(1)
myRecord.order_date = new Date()
myRecord.unit_price = 100
myRecord.freight = 20
```

Because this pointer is foundset-based, it is only valid as long as the foundset it "points to" persists, and the record it points to is still in the foundset. If the foundset is cleared or changed, or if the record pointed to is deleted, then the pointer will no longer work.

Another useful feature of getRecord() is that if you are dealing with a foundset that is displayed on a form, and you want to access a record in that foundset *without displaying the record*, getRecord() will allow you to achieve that. In a form-based foundset,

```
forms.myForm.controller.setSelectedIndex(10)
```

or

```
forms.myForm.foundset.setSelectedIndex(10)
```

will each cause the UI to *display* the 10th record. However,

```
forms.myForm.foundset.getRecord(10)
```

will return a pointer to the 10th record *without displaying it.* So let's say you need to scan all the records in a form's foundset, but you don't want the user to see the UI scroll madly as it displays each record. Using getRecord(n) to reach each record will achieve that result.

Another good use for the getRecord() function is to gain a pointer to a *new* record created using the newRecord() function. If you look at the documentation for the

newRecord() function, you will note that it returns an integer. This integer corresponds to the new record's index in the record's foundset. So if you say:

```
var x = foundset.newRecord()
```

then x will contain the index of the new record in the foundset. Used in conjunction with getRecord() you can do the following to both create a record and gain a pointer to it all in one line of code:

```
var myNewRecord = foundset.getRecord(foundset.newRecord())
```

It's also worth noting that these 'record pointers' that we've been talking about can be sent as parameters to your own methods, thus giving the called method access to that record, and only that record. So continuing from the example above, if I want to pass the record pointer to another method called populateNewRecord() I could do this:

```
populateNewRecord(myNewRecord)
```

Now in the populateNewRecord() method I can receive the record pointer just like any other argument by doing this:

```
var myRecord = arguments[0]
```

by Adrian McGilly, Servoy Consultant & Trainer

Assign a Foundset to a Variable

We've just seen that you can use getRecord() to assign a record pointer to a variable:

```
var myRecord = foundset.getRecord(10)
```

Similarly, you can assign an entire foundset to a variable. Take the example we used earlier where we were modifying a record in a related foundset that is reached via two nested relations:

```
foundset.customers_to_orders.orders_to_order_detail.setSelectedI
ndex(1)
// now the current record in this related foundset
// is the first record in the foundset
foundset.customers_to_orders.orders_to_order_detail.order_date =
new Date()
foundset.customers_to_orders.orders_to_order_detail.unit_price =
100
foundset.customers_to_orders.orders_to_order_detail.freight = 20
etc.
```

We could make this code less verbose by doing this:

```
var relatedFoundset =
foundset.customers_to_orders.orders_to_order_detail
relatedFoundset.setSelectedIndex(1)
relatedFoundset.order_date = new Date()
relatedFoundset.unit_price = 100
relatedFoundset.freight = 20
etc.
```

by Adrian McGilly, Servoy Consultant & Trainer

Dataproviders

by Adrian McGilly

Overview

It's important to understand the different kinds of containers that Servoy offers for storing and manipulating data and how to use them. The Servoy documentation does a good job of explaining how you create and use these containers, and this section goes into a bit more depth.

'Dataproviders' is the name Servoy gives to a group of data containers that can be placed on forms and can be used in relations. They include the following:

- database columns
- calcs
- stored calcs
- global variables (ones you create in the Solution Explorer's Globals/Variables node, NOT ones you declare directly in your JavaScript code)
- aggregations
- form variables

Let's look at each of these one by one:

Database Columns

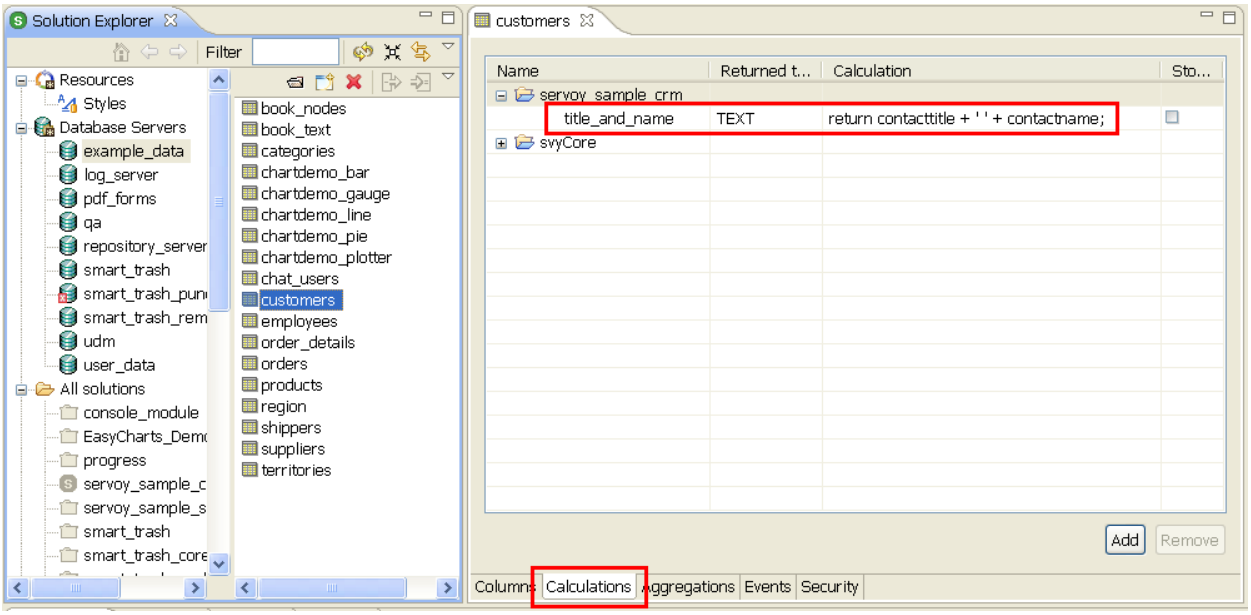
These are the tables and columns from the database. Please read the section entitled Where is my DB Schema Stored? for an understanding of how your Servoy solution is coupled with your database schema.

Calcs

Calcs are virtual, read-only columns whose values Servoy calculates based on a script your provide. You can use these dataproviders in your methods and display

them on forms but they are never stored to the db. For example in the illustration below the user has created a calc which puts the contact's title and name together into a single 'virtual column' called title_and_name. So on a record where contacttitle is 'Mr.' and contactname is 'Joe Smith' this calc will contain 'Mr. Joe Smith'.

Servoy evaluates calcs for each record automatically whenever a record is displayed on a form or made the current record in a foundset.



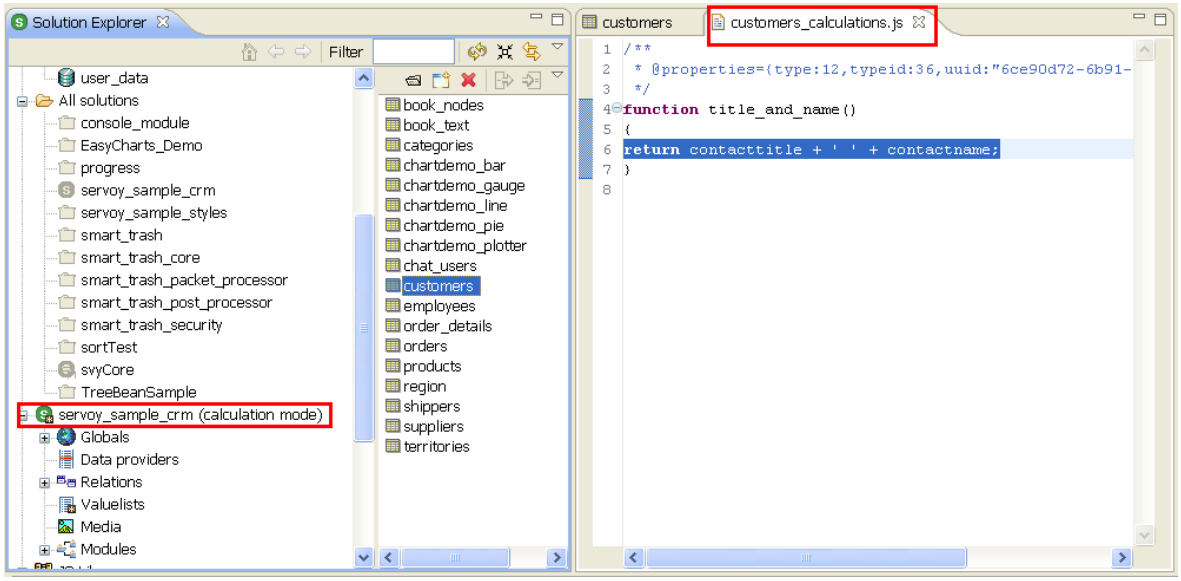
You create a calc by opening a db table from the Solution Explorer, then click on its 'Calculations' tab, then click on the 'Add' button and give your calc a name. If the name you give to your calc is identical to a db column in this table, then it will become a stored calc, which is described below. If not identical, it will become an unstored calc. This section deals with unstored calcs.

To tell Servoy what to store in a calc you need to use the calculation editor which you get to by double-clicking in the calculation column (where it says

```
"return contacttitle + ' ' + contactname;"
```

in the illustration above. You will notice that when you are in the calc editor (illustrated below), the Solution Explorer goes into 'calulcation mode' which excludes certain objects such as forms. Aside from Globals, the only dataproviders displayed in the tree are from the table that the calc is being defined on. Similarly,

it only shows relations that "start" with that table. Gone are all the forms, controllers, foundset objects, databasemanager objects and other objects you're used to seeing in the Servoy Editor.



There is a good reason for this.

Although Servoy doesn't stop you from writing code that reaches beyond the scope of that tree, it is not recommended you do so. It is recommended you stick to the things you can do with that tree. The reason for this is that calcs are evaluated more often than you might think, and putting expensive operations such as searches, updates to records, SQL commands, lengthy loops, etc. will degrade performance significantly.

Here are a couple examples of how you might use a stored calc:

Let's say you have a Customer table and you want to store a complex key for each customer based on zip code and the year the customer was entered in the system. So for a customer in zip code 94502 who was entered into the system in 1997 you want '94502-1997'. You could create a stored calc called customer_key of type char with the following method:

```
return zip_code + '-' + date_entered.getFullYear();
```

Now let's imagine that instead of storing that key in the customer table, you wanted to store it in the orders table. To do this, you would need a 'many-to-one' relation

called orders_to_customers that takes you from an order record to its parent customer record. Once that's in place, your calc method would look like this:

```
return orders_to_customers.zip_code + '-' +
orders_to_customers.date_entered.getFullYear();
```

Notice that when you create a calc, if your solution includes modules then Servoy makes you select which solution the calc will belong to. For instance in the illustration above, the user has chosen to create a the calc in the solution called servoy_sample_crm, not in the module called svyCore which is also listed. This means is that the calc will be performed whenever the servoy_sample_crm solution is accessing the customers table.

If the user had created a calc in the svyCore module, then anytime a solution that contains that module is running, it would perform that calculation.

Although calcs are usually read-only columns that return a scripted value, they can also be used as writeable virtual columns. If you calculation script reads simply as follows:

```
return;
```

then rather than Servoy calculating a value for the column, Servoy will let you put values in the column in your solution's methods or, if you place the calc in a field on a form (and make the field Editable) Servoy will let users enter values in the calc. So now you have a virtual database column that you or your users can manipulate, but that will never be stored in the database, and that will persist only as long as your foundset persists. This can sometimes be very useful, as demonstrated in this example of implementing checkboxes.

Stored Calcs

These are special calcs where the result returned by your script is stored in a database column in the db. You create a stored calc the same way as an unstored calc (see previous section) but make the name you give to your calc match a db column in this table. Then it will become a stored calc and you will se a checkbox in the 'stored' column.

You need to be mindful about how and where you use stored calcs.This is because calcs are evaluated everytime a record is displayed or accessed programmatically. If you place stored calc on a table whose records are displayed in table or list views, then when those forms are in use, Servoy will be doing a lot of calc work on a lot of records. Often I find that the stored calcs cause these records to go into a 'dirty' or edited-and-unsaved state when you don't expect them to, even if you have autoSave turned on, which can have several consequences such as:

- Your user will see the 'e' in the status area of his Servoy window, indicating that there is unsaved data, even if all of the user's edits have been saved, and this can be disconcerting for the user.
- Certain operations such as sorting won't work on a foundset that contains edited, unsaved records.

So when is it safe to use a stored calcs? I use them in solutions that don't have a UI, such as batch processors, and on tables that I know won't be displayed in list view or table view.

Aggregations

Aggregations are like unstored calcs that return aggregations (sum, average, count, maximum and minimum) on a foundset of records.

Let's say you are displaying a list of customers on a form. One of the columns in the customers table is called YTD_total containing the total value of orders placed so far this year by that customer. Let's say you want to display a total of all the YTD_total values in the current foundset on the form. Here are the steps:

Create an aggregation called sum_YTD_totals on the customer table,

Place a 'Trailing Grand Summary' section at the bottom of the form

Place the sum_YTD_totals aggregation in a field in the 'Trailing Grand Summary' section of the form. Presto! You're done.

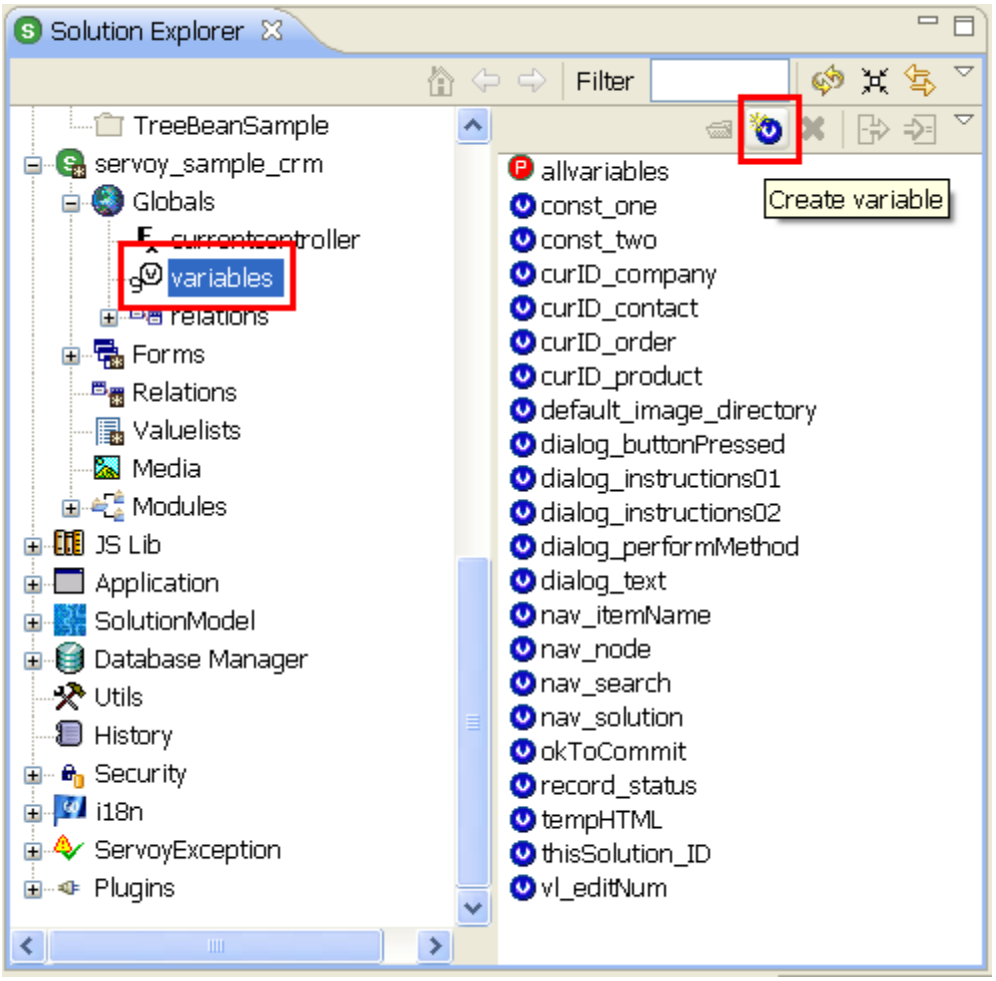
You can also use aggregations in conjunction with relations to aggregate across a group of related records without ever having to lookup or display said records. For example, let's build on our example above, and imagine that there's a table called salesreps, and there's a one-to-many relation between salesreps and customers called salesreps_to_customers. You could create a calc (stored or unstored, as you

wish) in the salesreps table that would calculate the total of all YTD_totals for all customers belonging to that salresrep. Here's the method you would use in your calc:

```
return salesreps_to_customers.sum_YTD_totals
```

Global Vars

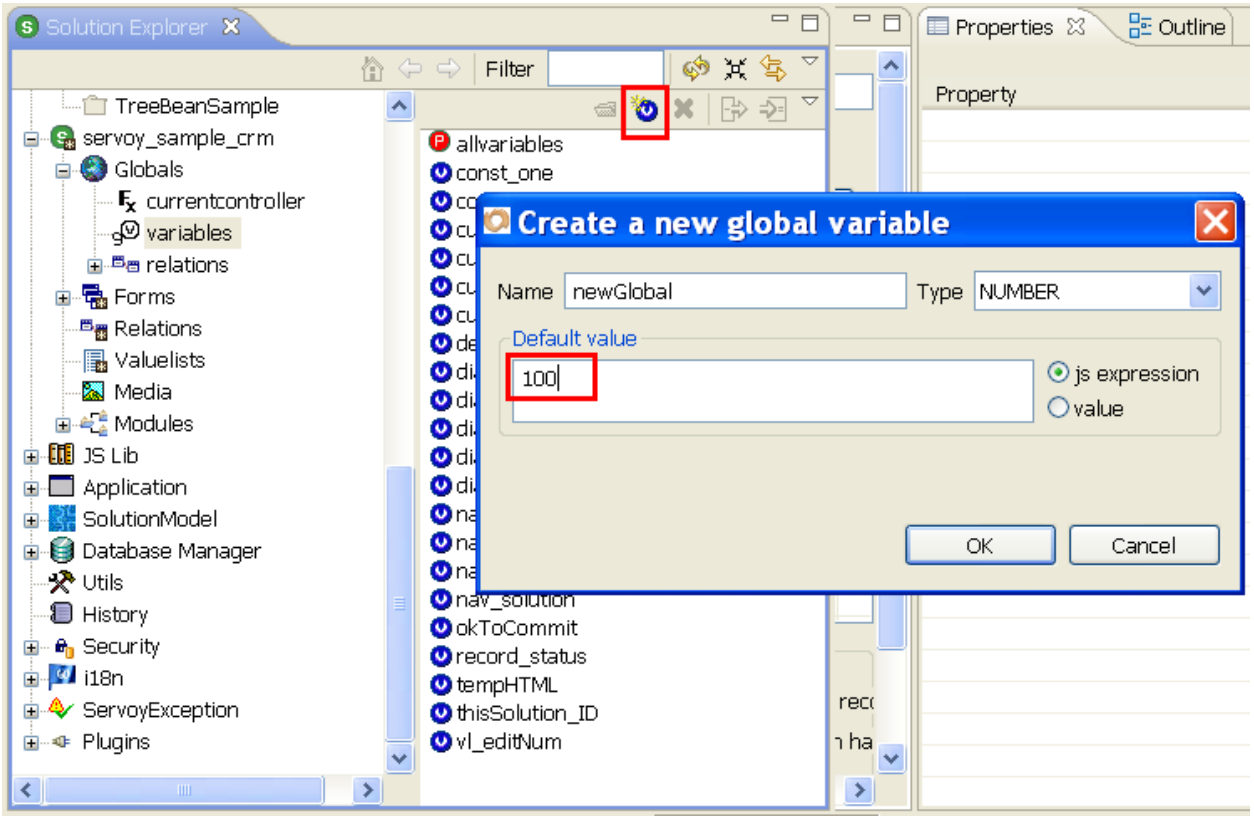
Global vars declared in the Dataprovider window are available to all your methods and everywhere dataproviders are called for (e.g. in fields on forms, in relation definitions, under the Globals>>Variables node of the object tree in the Servoy Editor, etc.).



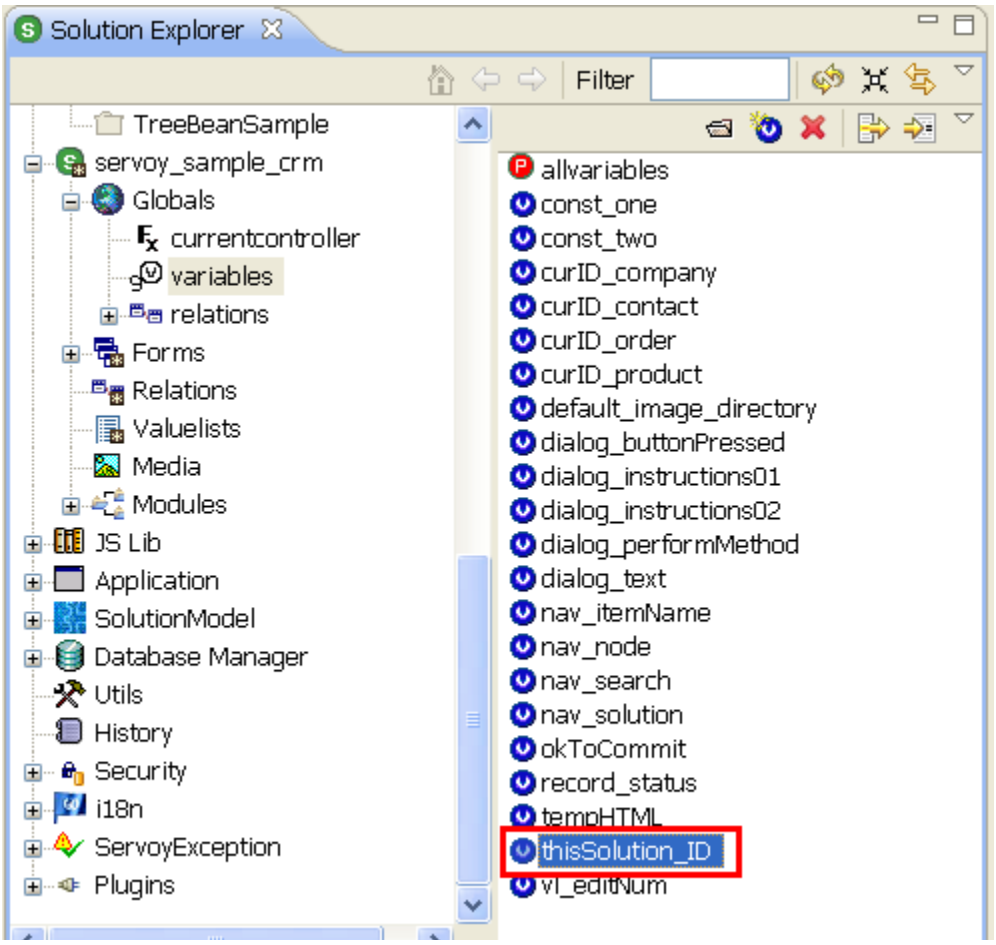
You refer to a global var by putting 'globals.' in front of its name, like this:

```
globals.last_name = 'Smith';
```

Global vars don't persist once you leave run-time mode. If you use ctrl-L to jump in and out of run-time mode while writing and debugging a method, they all revert to their default values which, unless you specify otherwise, is zero or empty (depending on the type of var). You can assign a default value to a global in the at the same time you create it:



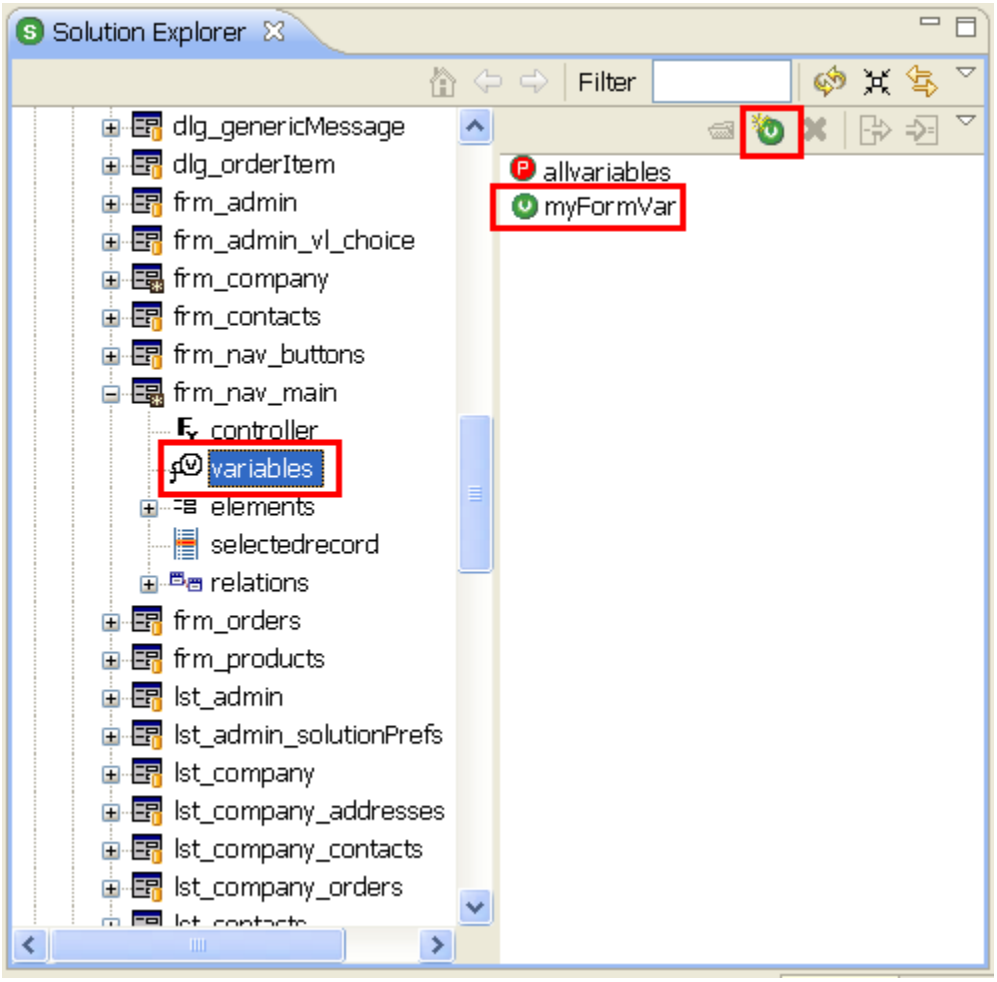
If you later want to change the default value you can double-click on the global variable in the solution explorer and change it in the script editor:


Solution Explorer
Filter
TreeBeanSample
servoy_sample_crm
Globals
currentcontroller
variables
relations
Forms
Relations
Valuelists
Media
Modules
JS Lib
Application
SolutionModel
Database Manager
Utils
History
Security
i18n
ServoyException
Plugins
allvariables
const_one
const_two
curID_company
curID_contact
curID_order
curID_product
default_image_directory
dialog_buttonPressed
dialog_instructions01
dialog_instructions02
dialog_performMethod
dialog_text
nav_itemName
nav_node
nav_search
nav_solution
okToCommit
record_status
tempHTML
thisSolution_ID
vl_editNum

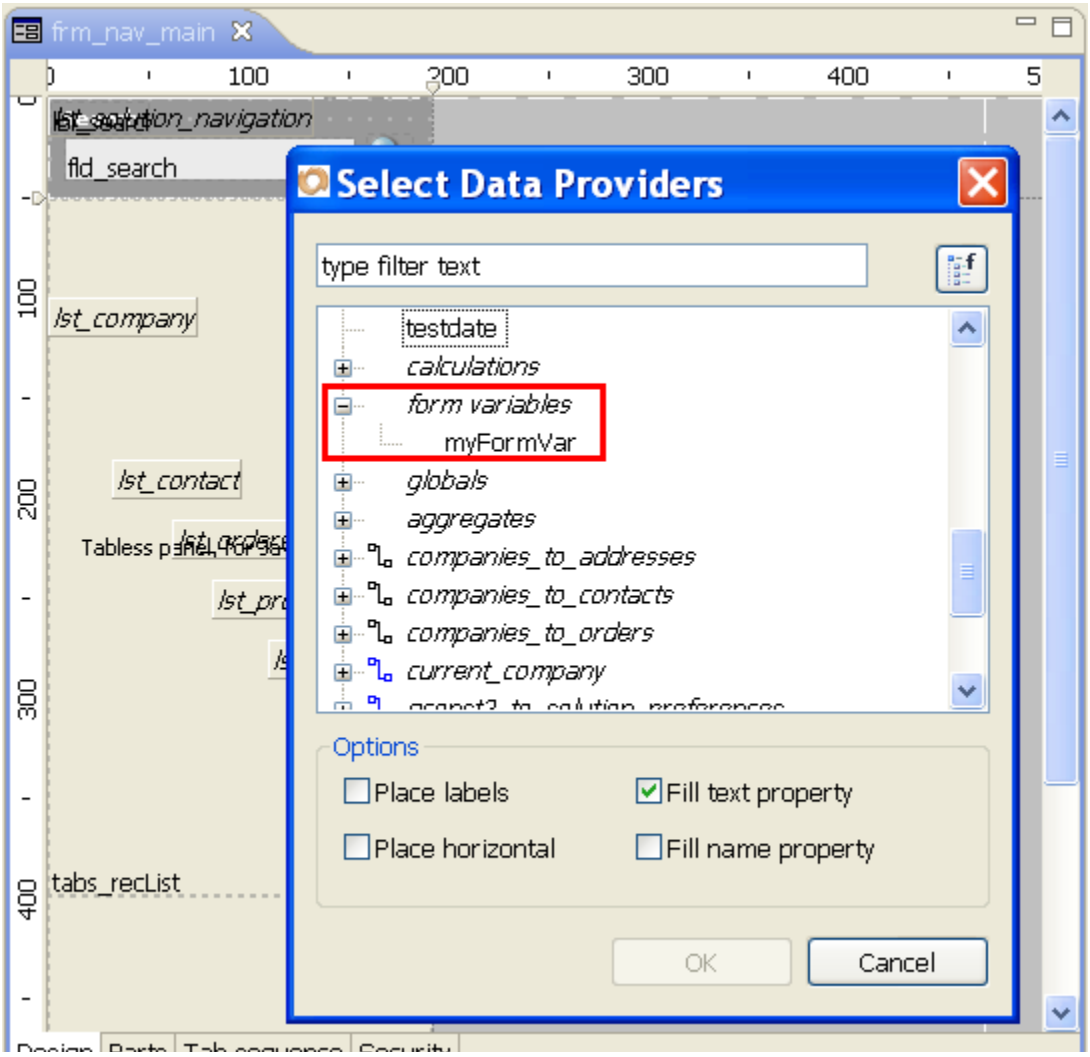
There are in fact two kinds of global variables possible in Servoy, the kind you declare in the Dataprovider widow, and the kind you declare directly in a JavaScript method. Global vars declared directly in your JavaScript code do not show up in the list of global vars in the Dataprovider window, nor do they show up in any lists of globals Servoy displays in its development environment. You can use them in your JavaScript code but you cannot place them on a form or refer to them in your relations. They are intended for more advanced JavaScript programmers and I recommend beginners not use them at first. (For more info on how to declare global variables in JavaScript read this article I wrote for Servoy Magazine).

Form Variables

As the name implies, Form Variables are variables whose scope is limited to the form they are defined on. They are created from, and listed under, the Form Variables node of the form they belong to:



A Form variable is considered a Dataproviders for the form it belongs to, and so it can be placed in a field, label, button on that form. Below is the dialog shown when the 'place field' button was clicked on the frm_nav_main form, which has a form variable called myFormVar.



How To Reference Dataproviders on Forms

It took me a little while to understand how you refer to dataproviders in forms & methods. I was used to referring to a database column like this:

```
database_table_name.column_name
```

but Servoy has me referring to the column like this:

```
forms.form_name.column_name
```

and

```
forms.form_name.relation_name.column_name
```

and my immediate reaction was to say 'But my column is defined in a table, not on a form! How do you know what table I'm referring to?' It turns out that because a form is always based on a table, by referring to the form, you are implicitly referring to its table. Furthermore, when you say

```
forms.customerForm.last_name
```

you are in fact referring to the last_name column of the *currently selected record* on the customerForm form. So for example if you wanted to set the last_name column to 'Smith' you would do this:

```
forms.customerForm.last_name = 'Smith'
```

and if you wanted to put the value of the last_name column in a global var you would do this:

```
globals.cust_last_name = forms.customerForm.last_name
```

The above examples are using the “forms.customerForm” prefix in front of last_name. If you are working on a method belonging to the customerForm form, then you could omit the “forms.customerForm” prefix everywhere in that method and it would work just fine. In other words, you could (and should) do this instead:

```
last_name = 'Smith'
globals.cust_last_name = last_name
```

The same syntax rules apply when you want to refer to calcs or aggregates.

How to Reference Dataproviders using Relations

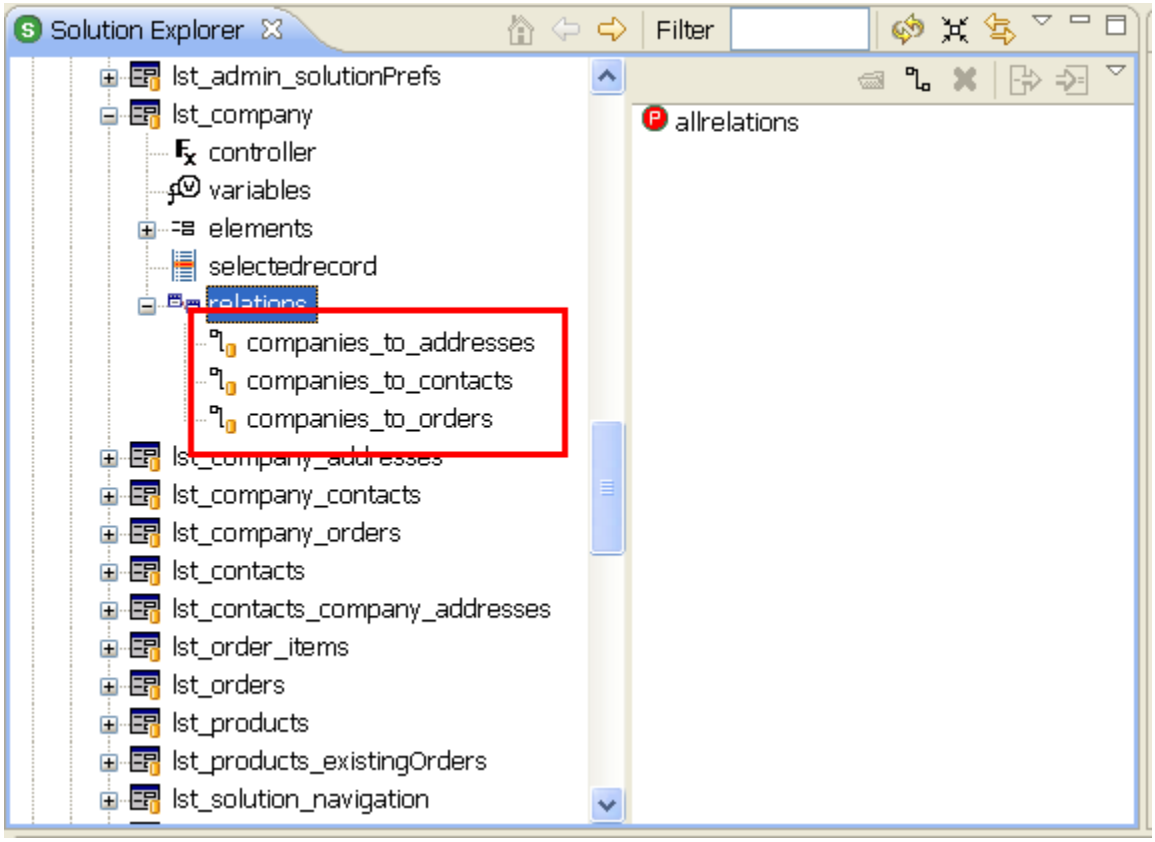
Overview

Now let's look at how you'd use a relation to refer to a dataprovider in a related record.

Every form is based on a table. In a sense, every *relation* is also 'based on' a table if you think of the table on the left hand side of the relation definition as its 'base table'. For example we can think of a relation called customers_to_orders as being 'based on' the customers table. The only

relations that aren't 'based on' a table in this way are global relations, which are 'based on' a global rather than a table. Let's leave those aside for now.

If you create several relations based on the customers table, such as customers_to_orders, customers_to_salesreps, customers_to_contacts, etc., these relations will all appear in the Servoy Editor under the relations node of any form that is based on the customer table. Below is an illustration showing that the form lst_company has three relations under its relations node, all of them having companies as their 'source' table.



You can use these relations to refer to dataproviders in the related table (i.e. the table on the right side of the relation) by using

```
relation_name.dataprovider_name
```

Many-to-One Relations

If the relation is a many-to-one relation (such as customers_to_salesreps, which takes you from a customer record to its assigned salresrep record) then by saying this:

```
forms.customerForm.customers_to_salesreps.last_name
```

you are saying 'the last name of the sales rep for the currently selected customer record on the customerForm.'

As with unrelated dataproviders, the 'forms.customerForm.' portion of these expressions isn't necessary if you are in a method that belongs to customerForm. So if you want to place the sales rep's last name on the customerForm, you will create a field and specify as its dataprovider:

```
customers_to_salesreps.last_name
```

and that would also be how you would refer to that dataprovider in a method belonging the customerForm form.

One-to-Many Relations

If the relation is a one-to-many (such as customers_to_orders, which takes you from a customer to the group of orders connected to that customer) then you will most likely use the relation in conjunction with a related tabpanel or perhaps a portal. You may want to skip ahead to the section on tabpanels to see how one-to-many relations are used with tabpanels, because the usage I discuss below is not typical. I am only including it for the sake of completeness.

When you say this:

```
forms.customerForm.customers_to_orders.order_date
```

you are saying 'the date of the currently selected order from the group of orders connected to the currently selected customer on the customerForm.'

You normally wouldn't see the above expression used as a dataprovider for a field on a form, but you might use it in your code. Let me explain.

If you put a field right on the customerForm whose dataprovider is:

```
customers_to_orders.order_date
```

it will indeed display the date of the 'currently selected order for the currently selected customer', but what is meant by the 'currently selected order'? If you're not displaying the list of related orders anywhere, your user

has no context by which to know which order is currently selected. By default, it will be the order with the smallest primary key among the orders related to that customer, but there is no context by which your user can see that.

To select the second order in the group of related orders you would issue this command in a method:

```
customers_to_orders.setSelectedIndex(2);
```

and as soon as you did that, the order_date from the second related order (as sorted by primary key) would appear on the form.

Although there are situations where this might be the desired interface, it is more common is to use a one-to-many relation to display columns from *all* the records connected to a parent, not just one record, and that's what related tabpanels (and portals) are for.

Using Related Dataproviders in TabPanels

Now let's imagine you want to *display* the order_date values of *all* the orders connected to a customer. You will create a related tabpanel on your customerForm based on the customers_to_orders relation. In the tabpanel you will place a form based on the Orders table, and on that form you will put a field with just 'order_date' as its dataprovider. This tells Servoy to display in the tabpanel all Orders records related to the current customer, and specifically to display the order_date column from those records. Notice that nowhere did you type

```
customers_to_orders.order_date
```

Instead, the customers_to_orders relation was specified in the tabpanel definition, and Servoy uses it to define its search when loading the tabpanel with Orders records. For a more detailed explanation of how to create related tabpanels, go here.

Using Related Dataproviders in Methods

Now let's say you have to write a method to loop through all the order_date values of all orders related to the currently selected customer, and return the oldest date. (Of course you could do this without writing any code using a

related aggregation, but let's ignore that for now.) In that context you *will* find yourself typing

```
customers_to_orders.order_date
```

Here's what the method would look like. This sample code assumes the method belongs to the customerForm form, and so I omit 'forms.customerForm' everywhere. It also assumes that you have confirmed *before* calling it that there is at least one order record related to the current customer (which you would do by checking that customers_to_orders.getSize() is greater than zero):

```
var most_recent = new Date(); /*this local var will hold
our most_recent order_date. This initializes it to the
current datetime.*/

for (var n = 1; n <= customers_to_orders.getSize(); n++)
{
     customers_to_orders.setSelectedIndex(n);/* this causes
the nth related orders record to become current, and loads
its column values into the related dataproviders.*/


     if (customers_to_orders.order_date <= most_recent)
     {
          most_recent = ustomers_to_orders.order_date;

     }
}

return most_recent;
```

Using Nested Relations and Dataproviders

In the section on nested relations, I explain how you can chain relations together to reach a distantly related dataprovider, like this:

```
relation1.relation2.relation3.dataprovider
```

Use the selectedrecord node to refer to Dataproviders

In the Servoy Editor, if you click on the “selectedrecord” node beneath any form in the object tree, you will see the list of dataproviders for the table that the form is based on. This will include database columns, stored calcs,

unstored calcs and aggregations. You can insert a reference to a dataprovider into your method simply by double-clicking on the dataprovider in this list.

Elements

When I started learning Servoy I had trouble understanding the difference between dataproviders and elements. This was largely because a dataprovider and an element can end up with the same name, and this lead to confusion as to which one I should use when. In case you've shared my confusion, this section will clear it up.

A **dataprovider** is a container that holds a piece of data. It can be in the form of a database column, a global variable, a calc or an aggregate. An **element** is a visual object on a form such as a field, a label, button, portal, tabpanel, javabean, a graphic, etc.. You must assign a name to a form element if you want to manipulate it in any way programmatically (e.g. control its visibility, size, color, position, whether or not it is editable, etc.)

Let's say you are working on a form called customerForm that is based on the customer table. When you add fields to the form using the field tool, it offers you the option of filling in the name property of the field elements being added to the form. If you select this option while adding a "last_name" field to your form, you would end up with a field element whose name is 'last_name' and whose dataprovider is also 'last_name'. This can get a little confusing.

What many developers do is to name their form elements manually by clicking on each one and typing a name into the name property on the property panel. (See the section on naming conventions for more info on this topic). You might want to use the prefix 'fld' when naming field element names (e.g. fldLastName and fldFirstName), a prefix of 'lbl' for label elements (e.g. lblLastName, lblFirstName) and 'btn' for buttons (e.g. btnLookupAccountBalance). Having done that, you would manipulate the field's properties like this:

```
forms.customerForm.elements.fldLastName.visible = true
forms.customerForm.elements.fldLastName.setFont('Arial')
```

but you would get or set the value of the last_name database column (or dataprovider, using Servoy's terminology) like this:

```
forms.customerForm.last_name = 'Smith'
// sets the last_name column in the current record to 'Smith'
```

```
globals.cust_last_name = forms.customerForm.last_name
/* puts the current record's last_name column into the
(predefined) global var globals.last_name.*/
```

The above examples are showing the "forms.customerForm" prefix just for clarity. If the method containing the above code is a form method on the customerForm form, then you could omit the "forms.customerForm" prefix everywhere and it would work just fine. In other words, you would manipulate the field's properties like this:

```
elements.fldLastName.visible = true
elements.fldLastName.setFont('Arial')
```

and you would get or set the value of the last_name database column like this:

```
last_name = 'Smith'
// sets the last_name column in the current record to 'Smith'

globals.cust_last_name = last_name
/* puts the current record's last_name value into the
(predefined) global var globals.last_name.*/
```

In the Servoy Editor, if you click on the "elements" node of any form, you will see the list of named elements on that form. When you click on an element, you will see its properties and functions listed below. You can insert a reference to an element's property or function into your method by simply double-clicking on the desired element or property in this list.

by Adrian McGilly, Servoy Consultant & Trainer

The Controller and Currentcontroller objects

by Adrian McGilly

Controller Object

Every form has a controller object. This isn't an object that you can see on your form, like a field or a button. It is invisible, except for its appearance in the Servoy Editor's object tree in every form's subtree. A form's controller object is a collection of properties and functions that act on that form and its foundset. Examples include:

```
controller.find()
controller.search()
controller.deleteRecord()
controller.setSelectedIndex()
etc.
```

Currentcontroller Object

currentcontroller just means 'the controller of the current form', and by 'current form' we mean the form that is currently visible in the main servoy window (not in a dialog window).

This object is useful when your code needs to do something to the form that is currently visible, but you don't know what form that will be at any given point in time. For example, let's say you're writing a global method that is in charge of printing the contents of the current form. You could use currentcontroller.print() and that would invoke the print() method of the currently visible form, no matter what it is.

Alternatively, let's say you have created your own print method called myPrint in each form of your solution, and suppose you want a global method that is supposed to call the myPrint method of the current form. How can you do that if you don't know what the current form will be at the time? You can use currentcontroller.getName() to get the name of the form, and from there you can build a call to its myPrint method. Here's how:

```
var formName = currentcontroller.getName()
forms[formName].myPrint()
```

(I am using the fact that plural group names in the object tree such as 'forms', 'elements', 'alldataproviders' can be treated as arrays. So forms.myForm is the same as forms['myForm'].)

I have explained how currentcontroller is used for the sake of completeness. In practice, I don't think it's wise to rely on this object too much in your code, precisely because it only works in the main Servoy window and not in dialog windows.

by Adrian McGilly, Servoy Consultant & Trainer

How & When Data Changes Are Saved

by Adrian McGilly

Servoy's Default Saving Behavior

In this section I will describe Servoy's *default* saving behavior. The next section covers how you can override that behavior and take more granular control of the saving functionality.

For now, I will assume you are not using database transactions. The situation where you are using transactions is discussed further down, but you should read this section first.

Let's say a user has just modified or added a record on a form. The default behavior of Servoy is to save changes or additions made by users as soon as they "leave" the record. It only performs a save if a record has been changed or added, not otherwise. This behavior is called autoSave.

User actions that will trigger an autoSave include:

- clicking on an empty part (background) of a form (Java Client only)
- clicking in or navigating to another record on the same form (in the case of list views and table views)
- adding another record
- deleting a record
- performing a search
- navigating to another form
- closing their solution
- logging out or leaving Servoy

As for data changes made by your methods, by default, the same rule applies. If your method makes a change to a record, that change will be saved to the db as soon as your method or the user does any of the above actions.

For example, if your method modifies record A and then selects record B using controller.setSelectedIndex(), your changes to record A are saved the moment record B is selected.

Consider this method:

```
controller.newRecord()
name = 'ABC Company'
city = 'Fremont'
return;
```

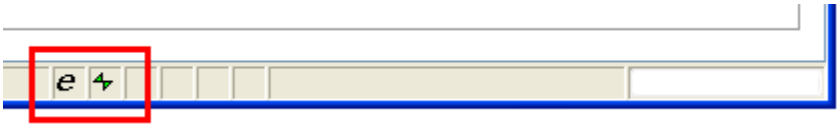
Immediately after this method runs, you will see 'ABC Company' and 'Fremont' in their respective fields on the form, but those values and indeed the new record itself are not saved to the db until either a method or a user performs one of the actions listed above.

Another example:

```
controller.newRecord()
name = 'Furtado Auto Supply'
city = 'Livermore'
controller.newRecord()
name = 'Honest Ed Auto Repair'
city = 'Larkspur'
return;
```

Immediately after this method runs, the 'Furtado Auto Supply' record is already saved to the db, but the 'Honest Ed' record, though it may be visible on the form, is not yet saved.

Servoy indicates that there are unsaved changes by displaying an 'e' in the status bar in the bottom right-hand corner of the window (Java Client only), as illustrated below. The other symbol you see below which looks like a bowtie indicates that there is a transaction currently active:



The other symbol next to the 'e' which looks like a bow-tie indicates that a transaction is pending. When all unsaved changes are saved, the 'e' goes away. When all pending transactions are closed (i.e. committed or rolled back) the "bow-tie" goes away.

Now let's look at deleting records. The controller.deleteRecord() is used to delete records, and it deletes from the foundset and the db immediately – no explicit saving or other action is required.

At any time, if you want to force Servoy to save changes to the db right then and there, you can do so by issuing one of these commands:

```
controller.saveData()
```

and

```
databaseManager.saveData()
```

As we discuss in more detail below, these commands force Servoy to push any unsaved changes to the db.

Overriding Servoy's Default Saving Behavior

If you want to have more control over when data is saved, you can turn off the autoSave feature by issuing the following command:

```
databaseManager.setAutoSave(false)
```

This tells Servoy to turn OFF the auto-saving behavior described above. (By default, when a solution is first created, autoSave is on.) With autoSave turned OFF any adds/edits/deletes will only be pushed to the database when you explicitly issue a saveData() command. This means it is possible for you to have several modified records and even several new records from one or more tables that are *all unsaved*. Because the data is unsaved, the 'e' in the status bar (illustrated below) will continue to appear until the records are saved.

If you are developing a solution and are curious to see this behavior in action, use your database administration tool (e.g. Sybase Central) to look at the raw data in your database as you make changes in Servoy, and you will see that the changes aren't reflected in the db until you issue the saveData() command.

The saveData() command comes in two flavors:

```
controller.saveData()
```

and

```
databaseManager.saveData()
```

They both do exactly the same thing: they push all unsaved adds/edits/deletes to the database. These unsaved operations might be on one table or they might span multiple tables depending on what you and/or the user have done since the last time data was saved.

Reverting Records Before Saving To The DB

There is a command which reverts any unsaved records to their 'original state', i.e. their current state in the db. This is useful if a user has edited a record (or records), and then for some reason decides to back out of the operation. That command is:

```
databaseManager.rollBackEditedRecords()
```

For example let's say you have a form where users enter/edit customer records. Before saving each record, you want to do a record-level validation, so you've turned autoSave off and you've provided 'Save' and 'Cancel' buttons on the form.

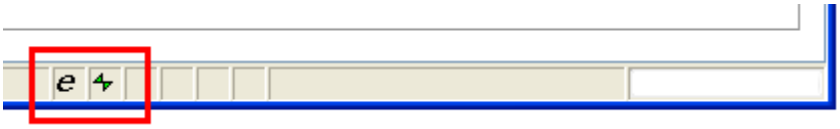
Now let's say your user has just edited an existing record and then they click the 'Cancel' button. Issuing the RollBackEditedRecords() command will revert the record to its original state prior to the edit.

How To Use Transactions

If you are using transactions, then everything I've said above applies, except that the changes made within a transaction aren't committed to the db until the transaction is committed. If you make multiple changes (e.g. add several records, delete several others, modify others) within one transaction, the changes will affect the foundset as they are made, but they will not affect the db until you commit.

Furthermore, if the transaction is rolled back, the changes are rolled back out of both the foundset and the db.

You can tell if there is currently an active transaction by looking at the status area at the bottom right of your running Servoy solution (Java client only). The symbol you see below which looks like a bowtie indicates that there is a transaction currently active . The 'e' indicates that there are unsaved changes to one or more record(s).



The commands for managing transactions are:

Command	**Description**
databasemanager.startTransaction()	Starts a new transaction
databasemanager.commitTransaction()	Commits the current transaction
databasemanager.rollbackTransaction()	Rolls back the transaction

by Adrian McGilly, Servoy Consultant & Trainer

Validating a Record Before Saving

by Adrian McGilly

Overview Of Data Validation

It is important to validate data entered by users before saving it to the db and Servoy offers comprehensive and flexible support for data validation.

Typically, there are two levels you might want to validate at – the field level (i.e. as the user is leaving the field) and the record level (i.e. as the user is leaving the record). Some situations call for field-level validation, some call for record-level validation, and some call for both. Record-level validation is necessary whenever the validity of one field depends on the contents of another, such as when a user is asked to specify the start and end dates of a date range.

Servoy lets you validate at both the field level and the record level.

There are a couple decisions to make before you start building validation logic into you solution:

Field-level validation: form-based or column-based?

Prior to v. 3.5, field-level validation could only be done by attaching validation methods to the onDataChange or onFocusLost events of fields on forms. If the same field appeared on ten forms always requiring the same validation, you still had to attach that validation method to the field's onDataChange event on all ten forms.

In v 3.5 Servoy introduced column-based validation, which allows you to attach a validation method directly to a database column. That validation logic will then be automatically applied to that column on *any form* on which it appears, and in *any solution* in the repository.

Record-level validation: form-based or table-based?

Prior to v. 3.5, record-level validation could only be done by attaching a validation method to certain form events such as onRecordSelection or onRecordEditStop (which was called onRecordSave in version prior to v 3.5).

In v 3.5 Servoy introduced table-based validation, which allows you to attach validation methods directly to a database table: one for insertions, one for edits and one for deletions. Those validation methods will automatically fire whenever a user attempts to add, edit or delete a row in that table on *any form in your solution.* (Note that unlike the column-based validations which apply to all solutions in the repository, these table-event-driven validation methods are solution specific.)

All of these approaches to validation are described in further detail below.

Field-Level Validation Using Field Events

For field-level validation, you can attach validation methods to these field events:

Event	**Description**
onDataChange	fires as the user leaves a field but only if the field was changed
onFocusLost	fires as the user leaves a field, regardless of whether or not it was changed

Usually validation is done on the onDataChange event. When an onDataChange event fires, it calls the method you've attached to the event and it sends it two parameters: the old value and the new value. You can grab these values as `arguments[0]` and `arguments[1]` respectively in your method.

You can then do your validation checks and if the validation fails you can display an error message. If you want to keep the user from leaving the field, simply return false from your method.

So let's say you wanted to write a validation method that checks if the value entered in the city field is 'Montreal', and if so, displays an error message and keeps the user from leaving the field. Here's what that method might look like:

```
if (city == 'Montreal')
{
     plugins.dialogs.showErrorDialog( 'Error',
                                'City can\'t be Montreal')
     return false;
}
```

Now let's say that in addition to that, after displaying the error message, you want the method to revert to the old value in the field. Here's what that method might look like:

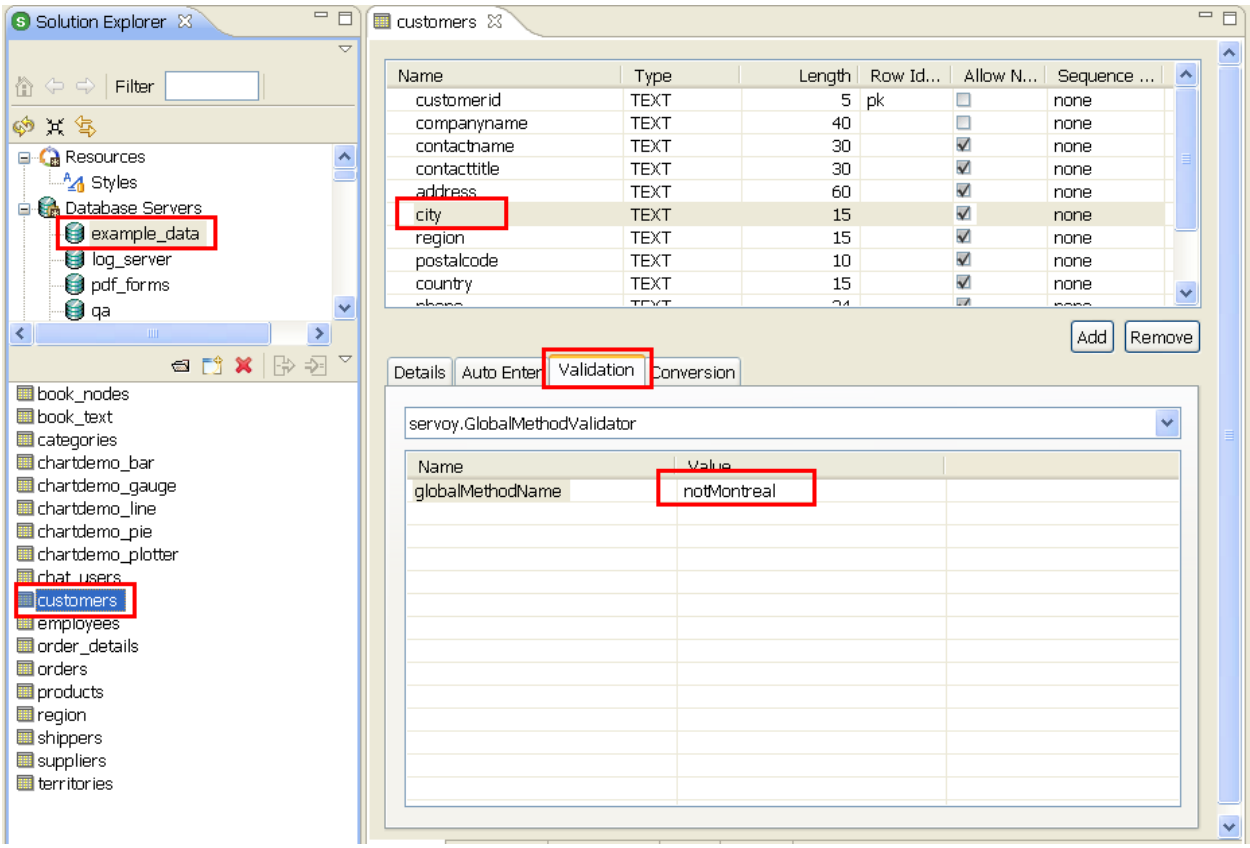
```
//get the old value
var oldValue = arguments[0]

//get the new value
var newValue = arguments[1]

if (newValue == 'Montreal')
{
     plugins.dialogs.showErrorDialog( 'Error',
                                'City can\'t be Montreal')
     city = oldValue
     return false;
}
```

The following limitation no longer exists as of v 3.5 - it is shown here in strikethru so that readers of past editions of this document who made development decisions based on this information know that it no longer applies:

Field-Level Validation: Column-Based

As of v. 3.5 Servoy introduced a new feature called *column-level validation*. For any database column you can assign a validation rule as a property of the column itself. You do this in the table editor by selecting the desired db column then select the Validation tab. The validation rule you specify there will be automatically applied to that column wherever it may show up *in any solution in the repository*. This last point is important: these validation rules are properties of the dataproviders themselves, and like all dataprovider properties, they apply across all solutions in your repository.

Below is an example of a column-level validation on the city column of the customers table in the db server called example_data:



Column-based validation is a great time-saving feature as it enables you to centralize all your validation logic in one place for all forms in all solutions. There are built-in validators for things like email addresses and numeric ranges, and these are all explained in the Servoy documentation. You can also create your own custom validators. What we will expand on here is the creation of custom validators and how they display error messages when validations fail.

To create your own custom column-based validator, you simply write a global method that performs the desired validation, returning true if the input is valid and false if not. Then you assign that method as the GlobalMethodValidator for the target column. In the example below, the global method notMontreal has been assigned to validate the city column of a table.

Now everytime a user enters data into the a city field, Servoy will invoke the notMontreal global method. If the method returns false, Servoy will prevent the user from leaving the field, otherwise Servoy will let the user leave the field. (You can also create custom validators based on Regular Expressions, using the RegExValidator, but we won't delve into that here.)

Let's look at an example. Let's say you wanted to make sure that the user never entered the value 'Montreal' into the *city* column. Here's what you would do:

1. Write a global method called *notMontreal* that checks if the user entered 'Montreal' into the city field, and if so, displays an error message. That method might look like this:

```
//the value entered by the user is passed
//in as arguments[0]
var city_to_validate = arguments[0]

if (city_to_validate == 'Montreal')
{
   return false
}
else
{
   return true
}
```

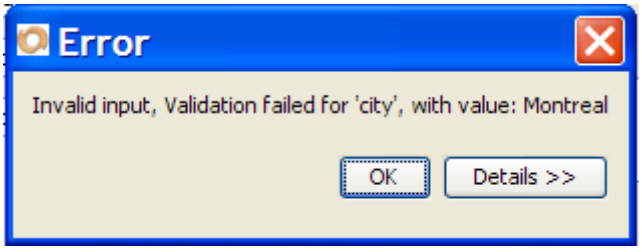
2. Go to the Database Servers node of the Solution Explorer and select the target database (example_data in the illustration above) and select the city column

3. Click on Properties then select the Validation tab

4. Select the servoy.GlobalMethodValidator validator type

5. Specify *notMontreal* as the globalMethodName as shown in the illustration above:

Now anytime the user enters 'Montreal' into the city field, regardless of what form he is on, he will receive this message:



and will be prevented from leaving the city field.

If you want to display your own message instead of this rather bland Servoy-generated dialog, have your validation method display a validation message before it returns false, such as in the example below:

```
//the value entered by the user is passed
//in as arguments[0]
var city_to_validate = arguments[0]

if (city_to_validate == 'Montreal')
{
     plugins.dialogs.showErrorDialog( 'Error',
                                'City can\'t be Montreal')
     return false
}
else
{
     return true
}
```

But wait - you're not done. Now your user will see your custom message followed by the Servoy-generated one. How do you suppress the Servoy-generated message?

That message is being generated by Servoy's error-handling system. To suppress it you need to create your own error-handling method and configure it to shut up when the error is related to invalid user input. Without getting into any details about Servoy error-handling in general, here is a quick recipe for suppressing that particular message:

1. Create a global method called errorHandler that looks like this:

```
var e = arguments[0];
if (e.isServoyException)
{
  if (e.getErrorCode() == ServoyException.INVALID_INPUT)
  {
       return;
   }
}
```

2. Go into File>>Solution Settings and specify this errorHandler method as the onError method.

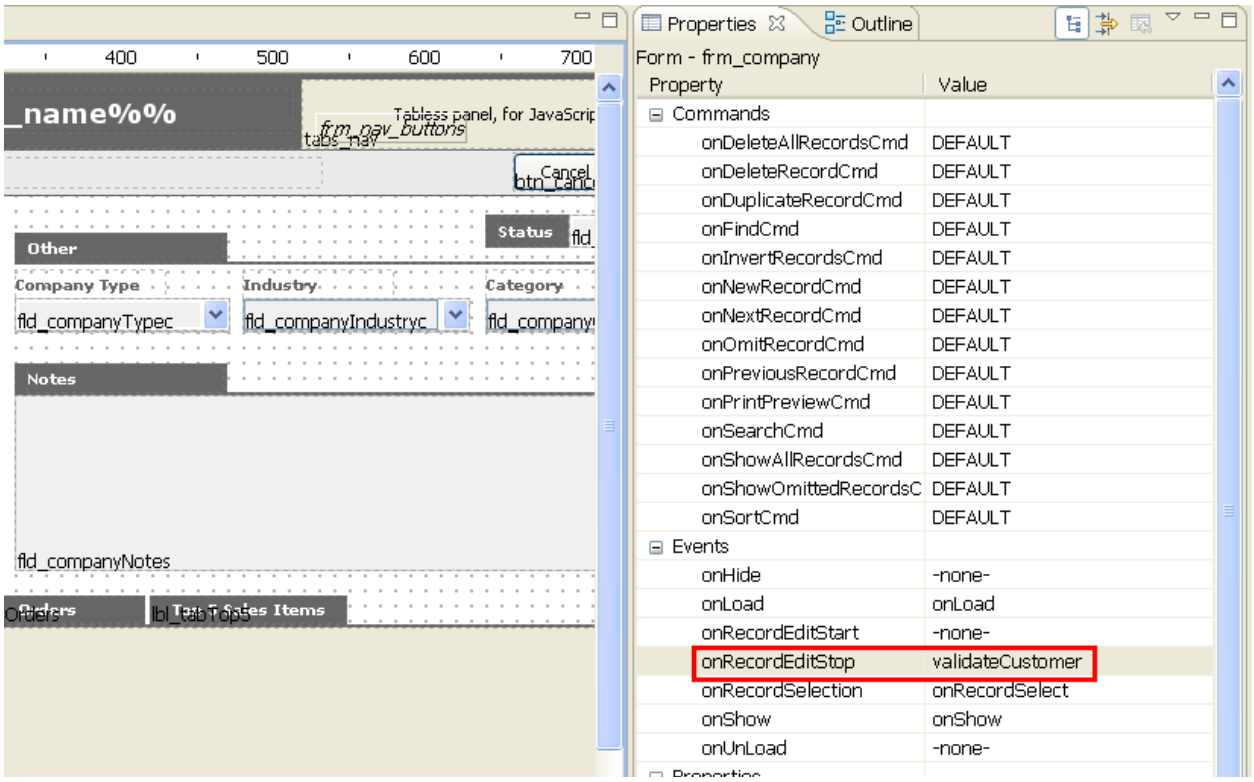
IMPORTANT NOTE: with column-based validations, anytime you edit the column *programatically*, the validation method will run. Any error messages you've programmed into your method will be displayed and if the method returns false, the edit will be reversed. It's important to keep this last point in mind before applying column-based validation, as there are times when we programmers need the flexibility to put certain values into fields programatically which users *aren't* allowed to put there. Column-based validation could get in the way of that, although there are ways to get around this problem if necessary. (A trick to get around this is to create a global variable, let's say 'doNotValidate', which you set to true when you want to temporarily disable table-event-driven validation, then check this variable within the validation method itself, bypassing the validation logic if it is set to true.)

WARNING: As mentioned earlier, these validation setings become properties of the dataproviders themselves and thus apply to that db column in all solutions that share the same repository. But in the case of GlobalMethodValidators, the global method that you specify resides in just one solution - the solution where it was created. If it gets renamed or deleted from that solution, it will no longer be available as a GlobalMethodValidator in that or any other solution, and that field will *always fail validation*. Similarly, if you delete the solution containing the global method, then the field will always fail validation in other solutions.

This is a good argument for putting these global validation methods in a separate module (see the section on using modules) that is designed to store resources that will be shared across multiple solutions, if only to make you more conscious of the fact that other solutions are dependent on these methods.

Record-Level Validation Using Form Events

Let's say you have record-level validations to perform on the customers table, but the validations vary depending on what form the user is in at the time. You will then want to do your validation at the form-level rather than use table events. To achieve this you will put your record-validation code in a method attached to the onRecordEditStop form event.



This is a *form event* and therefore it can be found in the properties panel when you click on the background of the form itself. It fires as the user is leaving a record. This applies regardless of HOW he is leaving the record, regardless of whether the record is being displayed in record view, list view or table view. The list of actions which trigger onRecordEditStop is the same as the list of actions which cause Servoy to perform a save when autoSave is turned on. That list can be found here.

The way you use this event for validation purposes differs depending on whther autoSave is ON or OFF.

autoSave ON

If autoSave is ON and you are doing record-level validation in an onRecordEditStop event, then you don't need to issue a saveData() command, but if you find the record is invalid you *can* tell Servoy NOT to save the record by returning false from the onRecordEditStop event.

autoSave OFF

If autoSave is off and you are doing record-level validation in an onRecordEditStop event, then your onRecordEditStop method should issue a databasemanager.saveData() command if it finds the record to be valid. If it finds the record to be invalid, it should inform the user of the problem and NOT issue a saveData() command.

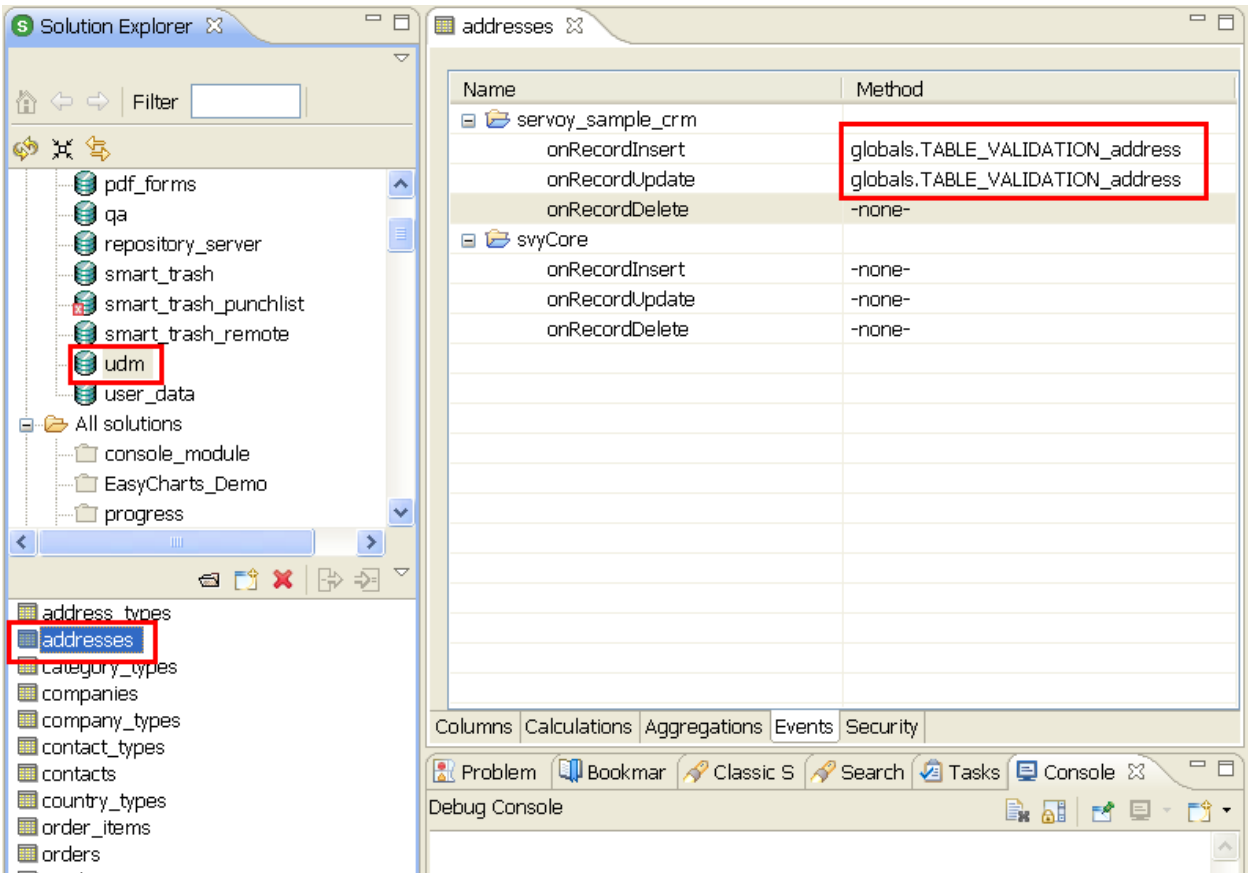
Record-Level Validation Using Table Events

In v. 3.5 Servoy added a feature called Table Events. There are three table events:

Event	Description
onRecordInsert	fires the moment Servoy is attempting to save a new record
onRecordUpdate	fires when Servoy is attempting to save edits to an existing record
onRecordDelete	fires when Servoy is attempting to delete a record

You can attach a method to each event and use it to take any actions you want. Servoy will send your method a pointer to the record being saved (or deleted) in the form of a record object (example below). With this record object you can perform any checks you need to perform and if you return false from the method, the event will not be allowed to complete, i.e. the record will NOT be inserted/updated/deleted.

There are many different ways you might want to use table events, but one obvious use is for data validation. Below is an example.



Based on the settings shown above, each time Servoy attempts to save a record to the addresses table, be it an existing record that's been edited or a new record, it will call the VALIDATION_address_record global method. (We could have specified different validation methods for onRecordInsert and onRecordUpdate but for simplicity we are using the same method for both cases.

Let's say we want the method to verify that the address1, city and zip_code fields are all filled, and if not, to display an error message and prevent the record from being saved. Here's what that method might look like:

```
//capture the current record object in a var
var record = arguments[0]

if (!record.address1 || !record.city || !record.zip_code)
{
     plugins.dialogs.showWarningDialog('Invalid input',
          'Address1, City and Zip Code are required fields');
     return false;
}
```

```
else
{
     return true;
}
```

Note that table events fire whether it is the user who is adding/editing/deleting a record or one of your methods that is doing it. So if you need the ability to make edits/insertions/deletions programmatically that your users aren't permitted to do, be careful about using table events. (A trick to get around this is to create a global variable, let's say 'doNotValidate', which you set to true when you want to temporarily disable table-event-driven validation, then check this variable within the validation method itself, bypassing the validation logic if it is set to true.)

Unlike column-based validation, table-events do not span multiple solutions, only the solution in which they are set up. Also, when a table-event method returns false, it doesn't invoke the Servoy error handler (as is the case with column-based validation), so the only error message that will be displayed is whatever messaging you program into your event method.

Controlling The UI For Data Validation Purposes

Above I've described how you can validate input before it gets saved to your db, but I haven't dealt much with the user interface, and how to restrict the user's movements in order to make sure he doesn't leave a record hanging in an invalid state. Usually when you detect that a user has entered invalid data, you will want to keep the user from doing anything else until he either corrects the data or cancels the operation in question. How can we achieve this sort of UI control in Servoy?

For field-level validations, the techniques described in the previous chapter are sufficient, because all we need to do is keep the user in the field until he enters valid data. Returning false from an onDataChange event or from a column-based validation method will achieve this.

For record-level validation, we still have some work to do. The techniques described above only ensure that if the user leaves a record in an invalid state, the record data won't be saved to the db, but they do nothing to restrict the user's movements. If the user edits a customer record, leaving it in an invalid state, and then clicks on a button that takes him to a different form, nothing we have done so far will stop him from going to that other form. Similarly if he has edited a

customer record in a list view or table view, leaving it in an invalid state, and clicks on a different customer record in the list, nothing is stopping him from going to that other record. This doesn't make for a very robust user interface. The UI should restrict the user's movements until he has either entered valid data or has confirmed that he wants to cancel the input.

When dealing with a list view or a table view, there is no way to keep the user from leaving one record and going to another, so if you have record-level validation to do on records in a list or table view, I recommend you first display the record in a separate form or in a dialog window.

Below are some techniques which can help you control the UI during validation:

Prevent the user from leaving a form until it's OK to do so

You can prevent a user from leaving a form by attaching a method to the form's onHide event. If this method returns false, the form will remain in place, even if the user is trying to go to a different form.

You want this method to return true if the data on the form is valid, and false if not. To achieve this you do the following:

1. Declare a global variable called, for example, globals.data_is_valid of type integer.
2. Initialize this var to 1 on the form's onShow event.
3. In your record-level validation method, set globals.data_is_valid to 1 if the data is valid, and 0 if not.
4. Have the form's onHide method simply return globals.data_is_valid. If the data is valid, it will return 1 (true) which will allow the form to hide, thus allowing the user to go to a different form. If not, it will return 0 (false), thus preventing the form from hiding, thereby keeping the user on the form.

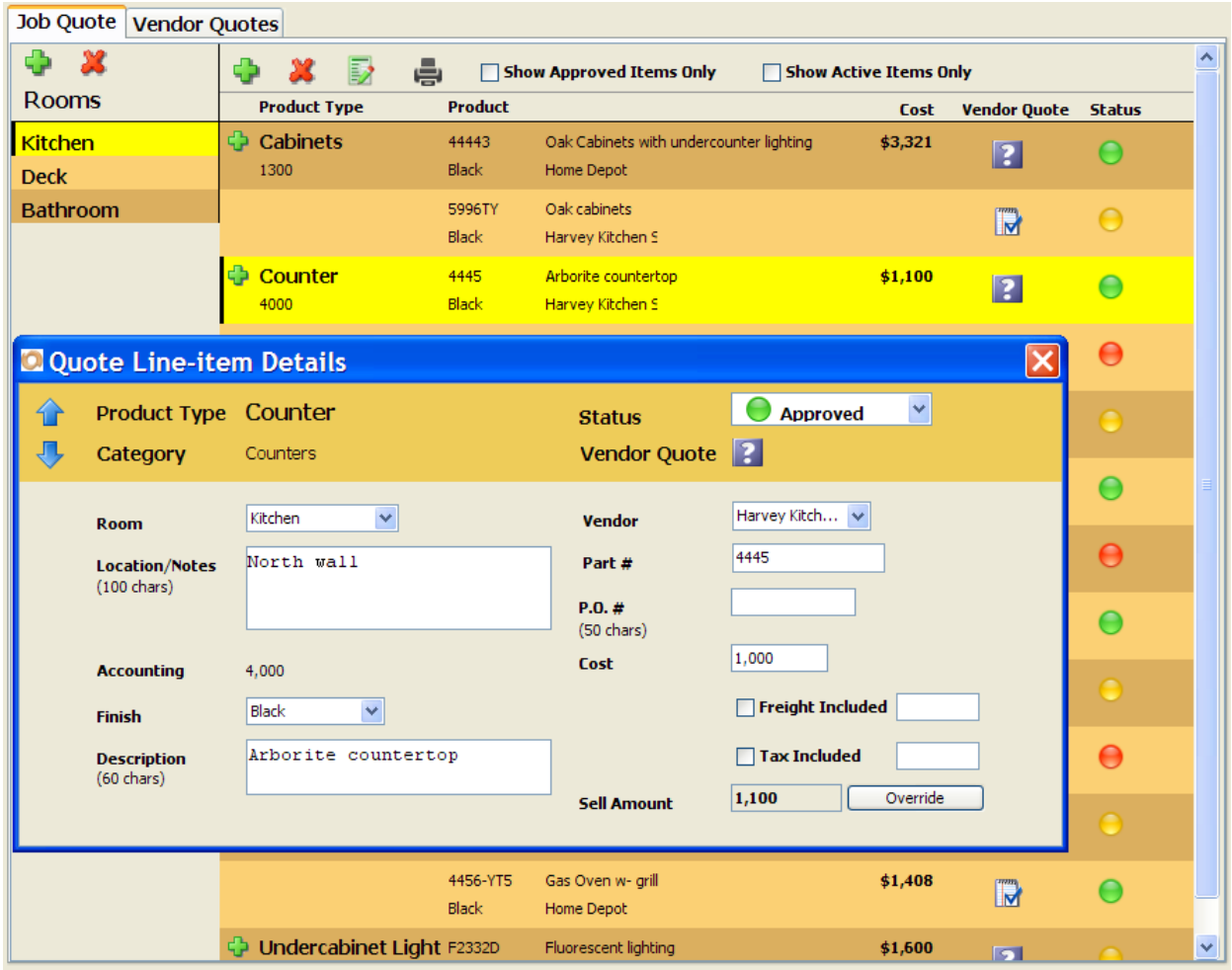
Do record edits and adds in a dialog window

You can do all your record editing in Record View in a dialog window.

```
application.showFormInDialog(forms.formName)
```

will display the specified form in a modal dialog window. So if the specified form contains just the record being added/edited, this approach keeps your user walled in during record editing and adding because he is restricted to the one record on the one form.

In the example below, a formInDialog (often abbreviated FID) is being used to display and edit the selected line-item in a job quote.



If the user closes the dialog window and the data he's entered isn't valid, before closing it you will want to ask the user something like "Data is not valid and can't be saved. Do you sill want to close the window".

The way to achieve this is exactly as described above in the section "Prevent the user from leaving a form". When the user clicks the close button on the

dialog window, it triggers the form's onHide event, and if that event returns false, the dialog window won't close.

Disable navigation controls until user is allowed to use them

Another approach is to do record editing/adding in Record View in a regular form, placing save & cancel buttons on the form (or on the navigator form, if you're using a navigator form) and make sure the user can't go anywhere else until the operation is either saved or cancelled. You will achieve this by disabling all navigation buttons & menu commands during the input operation. A good place to disable navigation buttons and enable your save and cancel buttons is in the onRecordEditStart event, which fires as soon as the user has started to edit a record or add a new one. Then re-enable them in the save & cancel buttons.

Here are a couple techniques to help achieve that:

1. You can disable an entire form all at once by setting the form's enabled property to false. For example:

   ```
   forms.customer.controller.enabled = false
   ```

   This can come in handy if you have a navigator form or a form in a tab-panel with navigation buttons on it and you want to disable all those buttons at once.

2. You'll also want to prevent the user from doing operations under the Select menu (such as Next, Previous, New Record, etc.) while editing or adding a record. The only way I know of preventing this to set the form properties corresponding to those commands (i.e. onNextRecordCmd onPreviousRecordCmd, onNewRecordCmd, , etc.) to your own set of methods, and for those methods to test whether a record is in an invalid state before proceeding with their normal course of action. For example, you could set the onNewRecordCmd to call a method called myNewRecordCmd, and this method would first test if the current record is in an invalid state. If so, it would warn the user and do nothing else. If not, it would issue create a new record using controller.newRecord.

by Adrian McGilly, Servoy Consultant & Trainer

Verifying a Record's Uniqueness

by Adrian McGilly

When adding or editing records, you will sometimes have situations where you want to verify that a certain column or group of columns are unique for that table. How do you do this in Servoy?

There are a number of ways of doing this.

If you don't mind your users receiving ugly error messages, you can set a uniqueness constraint on the table in your db and rely on that. In this case, when you save the record, your users will receive a beep and a rather unhelpful "Cannot Save Form Data" error messages. If the user clicks on the "Details" button beneath this message he will see a raw error message from the database citing a unique constraint violation. Not very pretty. Unfortunately in 2.2.5 there is no way to "intercept" this db message so as to present it nicely to the user. (That functionality is coming in 3.x)

The other approach is to verify the uniqueness by querying the database to see if there are any matching records before you attempt to save this new record. If so, you declare a uniqueness violation in whatever friendly way you want, and you return the user to the record without saving. If not, you save the record.

This sounds simple enough, but there is a catch. Let's say your user has added a new record on a form called customerForm, which is based on a table called Customer. He fills out its fields, putting 'ACME Tools' in the company_name field. Then he hits a save button. You've programmed the save button to verify the uniqueness of the company_name before saving the record. You might think you could do this:

```
//put the company_name from the new record into a local var
var co = company_name

controller.find();   // starts a find session
company_name = co;   // tells it to find on company_name = co
controller.search();  /* actually performs the search and puts
the results in the current foundset */
if (controller.getMaxRecordIndex() )
{
```

```
     plugin.dialog.showErrorDialog('Uniqueness Violation',
'There is already a company with the name ' + co, 'OK');
}
else
{
     controller.saveData();
}
```

The reason this won't work is that in the process of searching for a matching record, you have wiped out the foundset which was previously held by the form, and that foundset included your new record. When your method reaches the saveData() command, your new record is gone.

The same problem would arise if the user had edited an existing record, and you wanted to verify the uniqueness of the modified record.

Now, sticking with the example above (checking uniqueness of column company_name in table Customer on form customerForm) I am going to show you a couple different ways to check uniqueness without disrupting your foundset:

Approach #1 – Use A Global Relation To Test Uniqueness

You can create a global relation to test for uniqueness. This spares you writing any SQL, but the downside is you'll have to create one such relation for each column you wish to test for uniqueness. (Relations are discussed in greater detail in the chapter called Relations Explained – you might want to read that first and then come back here.)

In our example, here's what this would look like:

1. Create a global var called testValue.
2. Create a relation called testValue_to_company_name
3. Set the primary key (left side) to globals.testValue
4. Set the foreign key (right hand side) to the company_name column from the customer table

You now have a relation that points to all records in the Customer table whose company_name column matches the value in globals.testValue. You can use this to

test the uniqueness of that value by testing the 'size' of the relation's foundset using the getSize() function.

Returning to our example: your user has just entered a new Customer record and hit save. You want to test the uniqueness of the company_name field. Here's your method:

```
if (testValue_to_company_name.getSize() == 0)
{
     // record is unique
     controller.saveData
}
else
{
     // record is not unique. Display error msg, etc.
}
```

You can of course re-use globals.testValue in any other global relations you need to create for testing uniqueness of other columns.

Approach #2 – Use A JSFoundSet Object

In the section on JSFoundSet objects I described how you can create a foundset 'on the fly' in code and use the JSFoundSet functions to perform the same db operations that a form's controller allows you to do, plus a few extra functions.

This provides yet another approach to verifying a record's uniqueness. Using the same example, here's how you would do this:

```
// turn off autoSave, otherwise as soon as we hit the find()
// function three lines down, a save will be triggered and the
// record will already be in the db
databaseManager.setAutoSave(false)

var fs = databaseManager.getFoundSet('example_data',
'customers')

fs.find()
fs.companyname = companyname
var result = fs.search()

if (result == 0)
{
     /* record is unique, so save it using the controller from
```

```
the currently shown form, customerForm*/

     controller.saveData
}
else
{
     // record is not unique - display error msg, etc.
}

// turn autoSave back on
databaseManager.setAutoSave(true)
```

Sidenote about turning autoSave on and off: rather than turn autoSave off and on again, the safer way to do this would be to find out what the status of autoSave was BEFORE the uniqueness check, and restore it to that same status AFTER. The databaseManager.getAutoSave() function returns the current state of autoSave. Armed with that, you could do this BEFORE the uniqueness check:

```
var autoSaveStatusBefore = databaseManager.getAutoSave()
databaseManager.setAutoSave(false)
```

and then do the following AFTER the uniquess check to restore autoSave to whatever status it had before you turned it off:

```
databaseManager.setAutoSave(autoSaveStatusBefore)
```

Approach #3 – Make The Database Server Verify Uniqueness

As of version 3.0 Servoy enables you to trap error messages generated by the back-end database. This means it is possible to create an error handler in Servoy which, among other things, can trap for a message coming back from the db saying "the record you just tried to save violated a uniqueness constraint". It is beyond the scope of this handbook to delve into how exactly you would go about doing this, but here are two pointers.

1. Create an global method for error handling and in the Solution Settings (under the File menu) select that method as your On Error method. Now anytime Servoy encounters a run-time error this method will be called.

2. In the Servoy Editor, open this method and navigate to the ServoyException node of the Solution Object Model tree. Select that node and hit the "move sample code" button. This moves some sample error handling code into your

method. By studying this code you should be able to figure out how to trap for error messages coming from your database. Note that every db vendor has its own set of error codes so you will have to become familiar with the codes returned by the db you are using.

Approach #4 – Write A Generic Checkuniqueness() Function Using SQL

You can use an SQL query to determine uniqueness. SQL Queries sent with the databaseManager.getDataSetByQuery() function don't interfere with foundsets (unless you ask them to!).

databaseManager.getDataSetByQuery() lets you send a SQL query to your db and get back the result as a dataset. By default, this command does not affect any foundsets in any way so it will work nicely in this situation.

Let's say you have a form called customerForm with a form method called "Save" that is supposed to test for uniqueness of company_name in the Customer table. If it is unique, it should save the record, and if not it should bring up an appropriate error message. In the next example we use a SQL function called count(*).The count(*) function is a standard SQL function for counting all rows that match the criteria in the where clause. Here's what the method might look like:

```
/* get a count of the number of rows with the same company_name
value.
This query will return a single number, so the resulting dataset
will be a 1 x 1 array.
Note the use of the '\' escape character to insert
quote characters into the query.
*/

var query = 'select count(*) from Customer where company_name =
\'' + company_name + '\'';
var resultDataSet = databaseManager.getDataSetByQuery(
controller.getServerName(), query, null, 1);

/* the getValue function takes as its params a row number and a
column number and returns
the value in that cell of the dataset. If that equals zero, we
know there was no matching
record in the db so the record being created is unique.
*/
```

```
if (resultDataSet.getValue(1,1) == 0)   // test the number of
rows counted
{
     //record is unique
     controller.saveData();
}
else
{
     plugins.dialogs.showErrorDialog('Error', 'A customer with
company name ' + company_name + ' already exists.', 'OK');
}
```

Now you might be thinking "that's an awful lot of code just to test the uniqueness of a record!" and you'd be right. But you can reduce this to a one-line function call by putting all this code into a global function which you could call "checkUniqueness". Once you do that, your uniqueness test will be as simple as this:

```
if (globals.checkUniqueness('Customer', 'company_name',
company_name))
{
     controller.saveData();
}
else
{
     // show your error dialog here
}
```

This function would take as parameters the table name, column name and data value to be tested, and return true if the record is unique, and false if not. Here's what this function would look like:

```
/**************************************

FUNCTION: globals.checkUniqueness()

PURPOSE: checks whether a record is unique for a given table.
Returns true if so, false if not

PARAMETERS:
1. table name,
2. name of column that must be unique,
3. value to test for uniqueness
```

```
SAMPLE CALL:

The following sample code returns true if there are no rows in
the client table with 'Smith' in the name column:

globals.checkUniqueness('clients', 'name', 'Smith')

**************************************/

var tableName = arguments[0]
var columnName = arguments[1]
var columnValue = arguments[2]

/* In the next two lines of code we make use of a feature of the
getDataSetByQuery function which makes it easier to bind column
names to literal values in your WHERE clause by putting question
marks where you want your literal(s) to appear, and sending the
literal(s) as the third parameter of the getDataSetByQuery
function. This parameter must be an array, which is why we put
columnName in [] - this creates an array with columnName as its
only member. The first parameter of the getDataSetByQuery call
is currentcontroller.getServerName() which returns the name of
the db server that the current form is based on.*/

var query = 'select count(*) from ' + tableName + ' where ' +
columnName + ' = ?';
var resultDataSet = databaseManager.getDataSetByQuery(
currentcontroller.getServerName(), query, [columnName], 1)

if (resultDataSet.getValue(1,1) == 0)   // returns the value in
the single-cell dataset. If that equals zero, we know there was
no matching record in the db so the record being created is
unique.
{
     return true;
}
else
{
     return false;
}
```

If you wanted to get fancy, you could enhance this function to support multiple-column uniqueness constraints by allowing the call to send multiple column names

and multiple column values each held in an array. Something to try in your spare time!

Approach #5 – Use A Separate Form's Foundset

(The following approach will work in 3.x with autoSave turned off. It fails in 2.2.5 because it triggers a save that bypasses your validation rules.)

If for some reason you are afraid of SQL and the JSFoundSet object and you just want to do everything using forms and controllers, you can do so as long as you do it in a way that doesn't disrupt the foundset of the form containing your new record. You could, for example, create a new form based on the Customer table, turn on its *useSeparateFoundset* property and perform the find()/search() using that form's controller. Here's how that would look:

The form on which the new record has been entered is called customerForm, and it is the form currently shown. The form on which you'll perform the find()/search() to determine uniqueness we will call customerLookupForm. Below is a form method on the customerForm form for testing uniqueness of the company_name field:

```
var autoSaveStatusBefore = databaseManager.getAutoSave()

databaseManager.setAutoSave(false)

var co = company_name

customerLookupForm.controller.find()
customerLookupForm.company_name = co
//search() returns number of records found
var result = customerLookupForm.controller.search()

if (result == 0)
{
     /* record is unique, so save it using the controller from
the currently shown form, customerForm*/

     controller.saveData
}
else
{
     // record is not unique - display error msg, etc.
}
```

```
// restore autoSave status to its original status
databaseManager.setAutoSave(autoSaveStatusBefore)
```

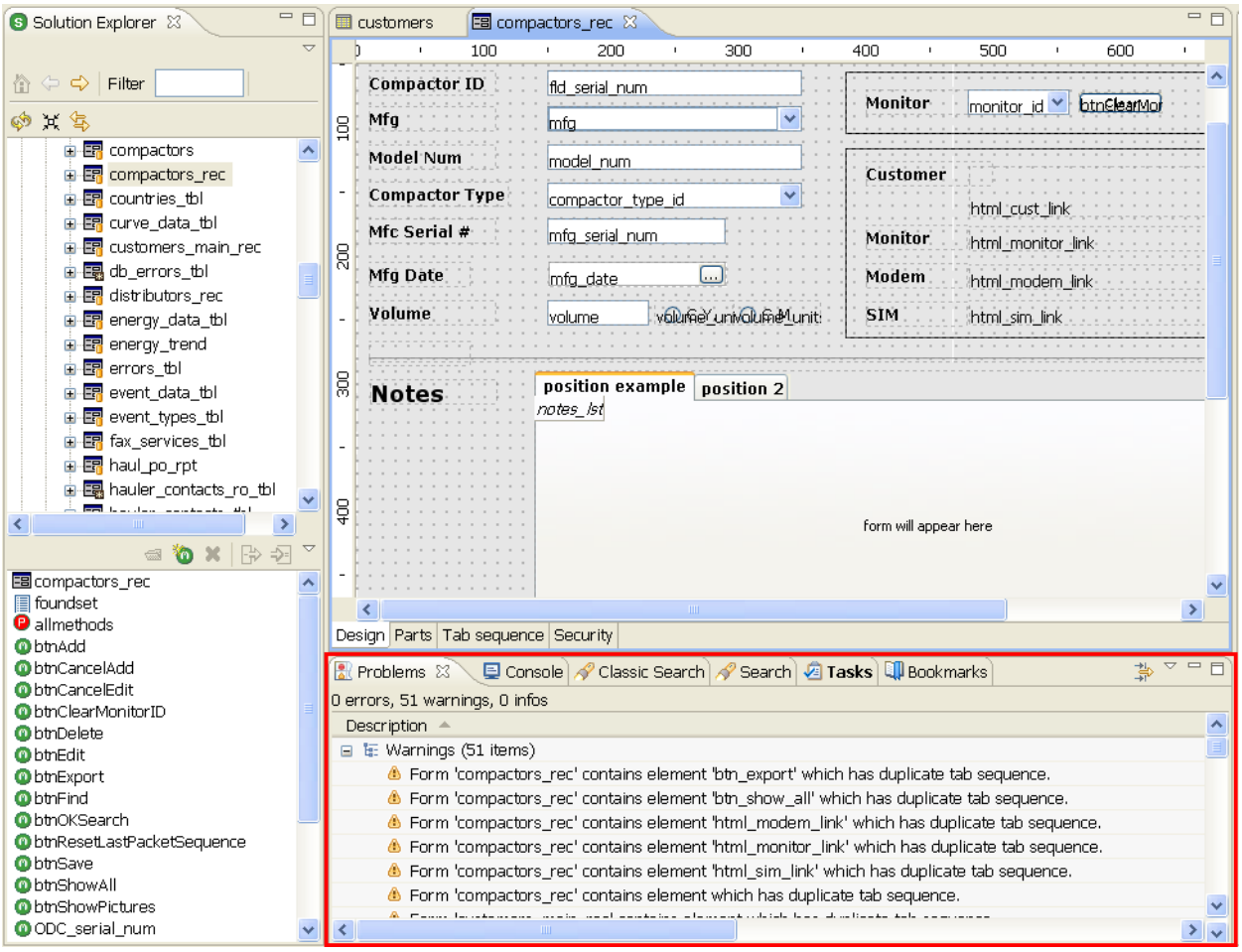
by Adrian McGilly, Servoy Consultant & Trainer

Error Handling in Servoy

by Adrian McGilly

Default Error Reporting by Servoy

It can be a bit tricky to figure out how and where Servoy logs run-time errors that occur in a running Servoy solution, and how you can intercept those errors and handle them 'elegantly'. This section does not deal with design-time errors, which Servoy reports in the 'Problems' pane of the developer IDE as shown below.

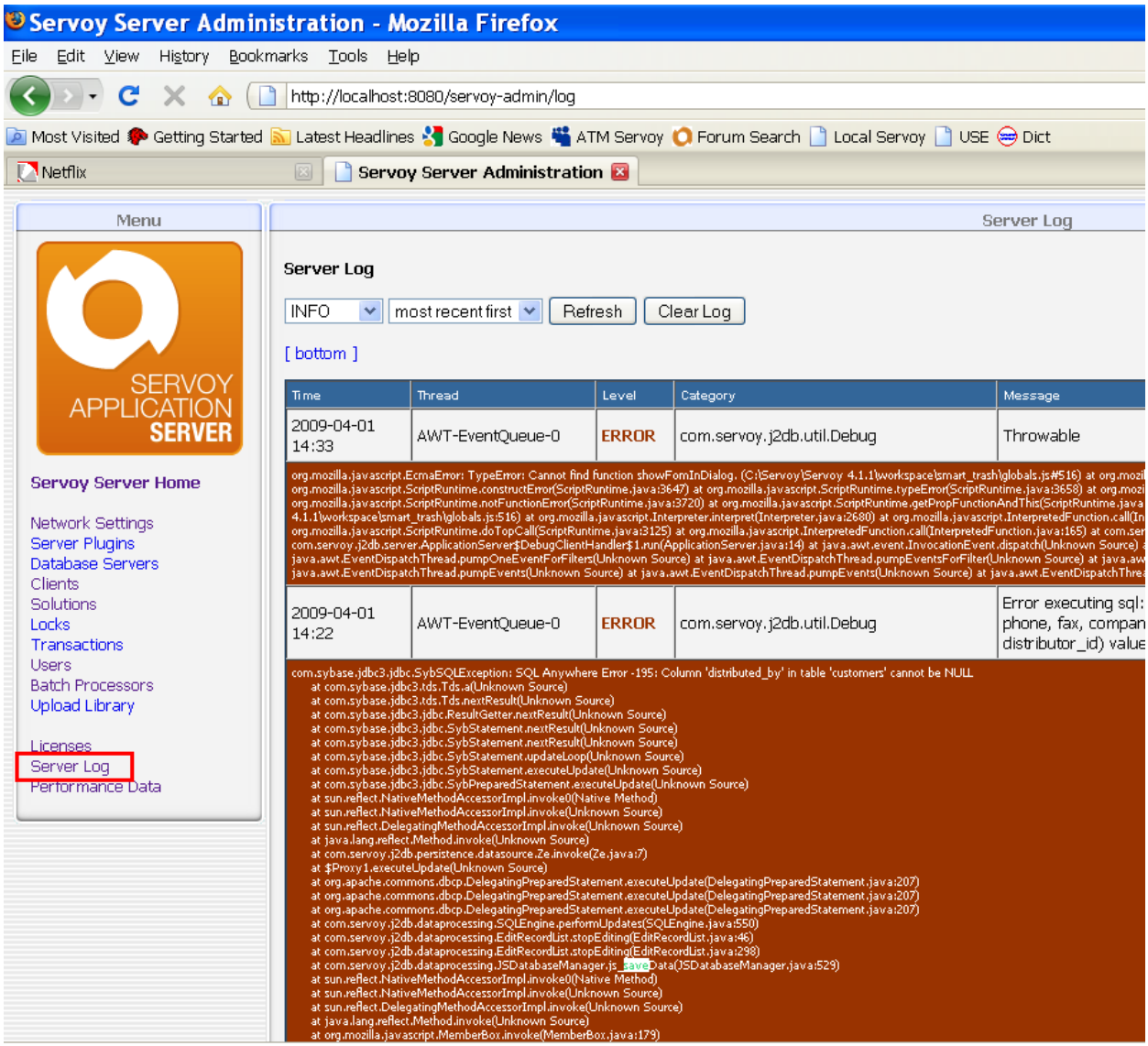


There are two main categories of run-time errors you need to be concerned with:

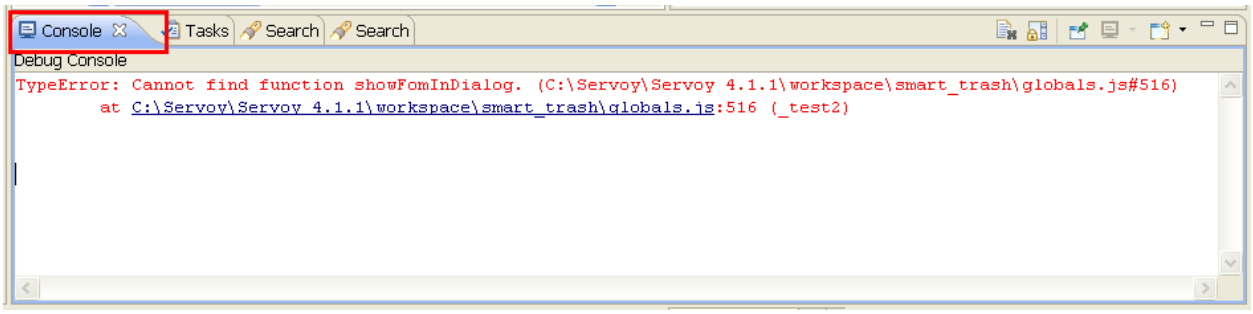
- Programming errors

- Database errors

Whether a solution is running in Servoy Developer or in a true Servoy client, by default all errors get logged in a file called application_server/servoy_log.txt. Each error includes a timestamp, a description of the error and an error stack. This is a detailed list of methods that were running at the time the error occured, which can be helpful to Servoy Support staff when diagnosing the cause of an error. They also appear in the browser-based Admin Console under the 'Server Log' section as shown below:



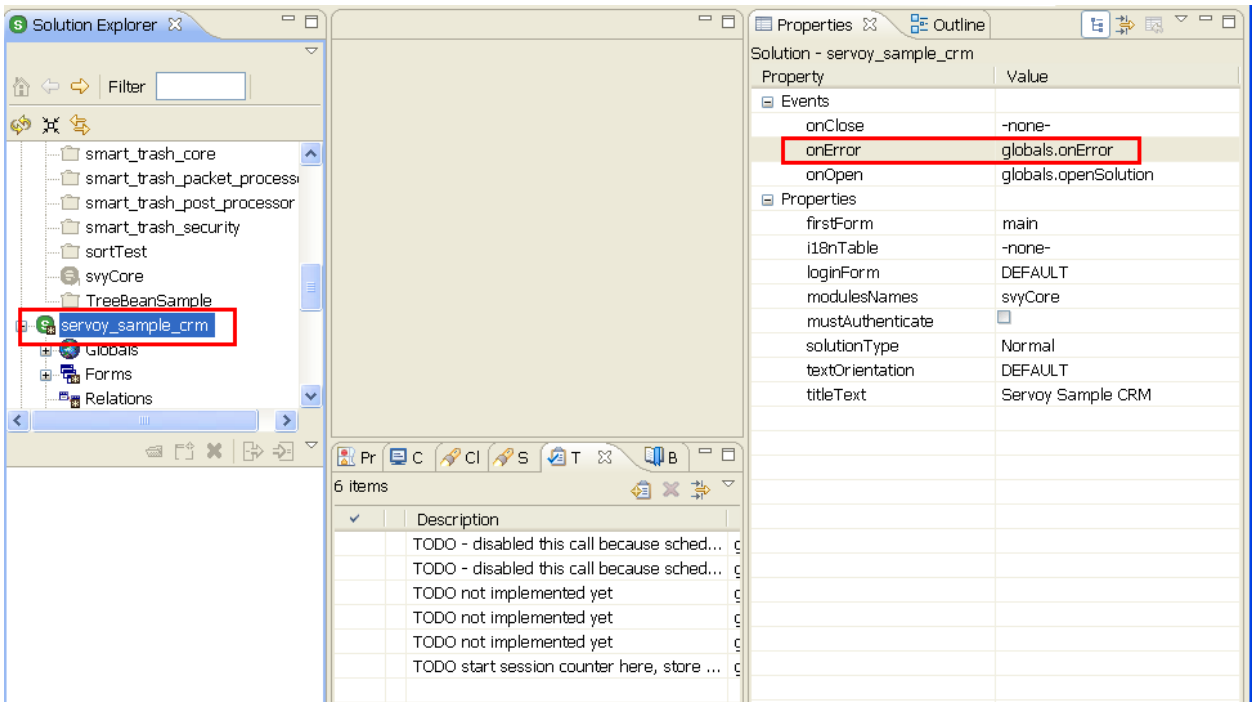
If the solution is running in Developer and you haven't implemented any error handling, then errors also get reported to the Console tab of the IDE (shown below).



However database errors (e.g. not-null constraint violations, invalid foreign key assignments, uniqueness constraint violations) do not get reported to the console, and are therefore a bit harder to detect. In the next section we discuss how to detect and handle programming and database errors.

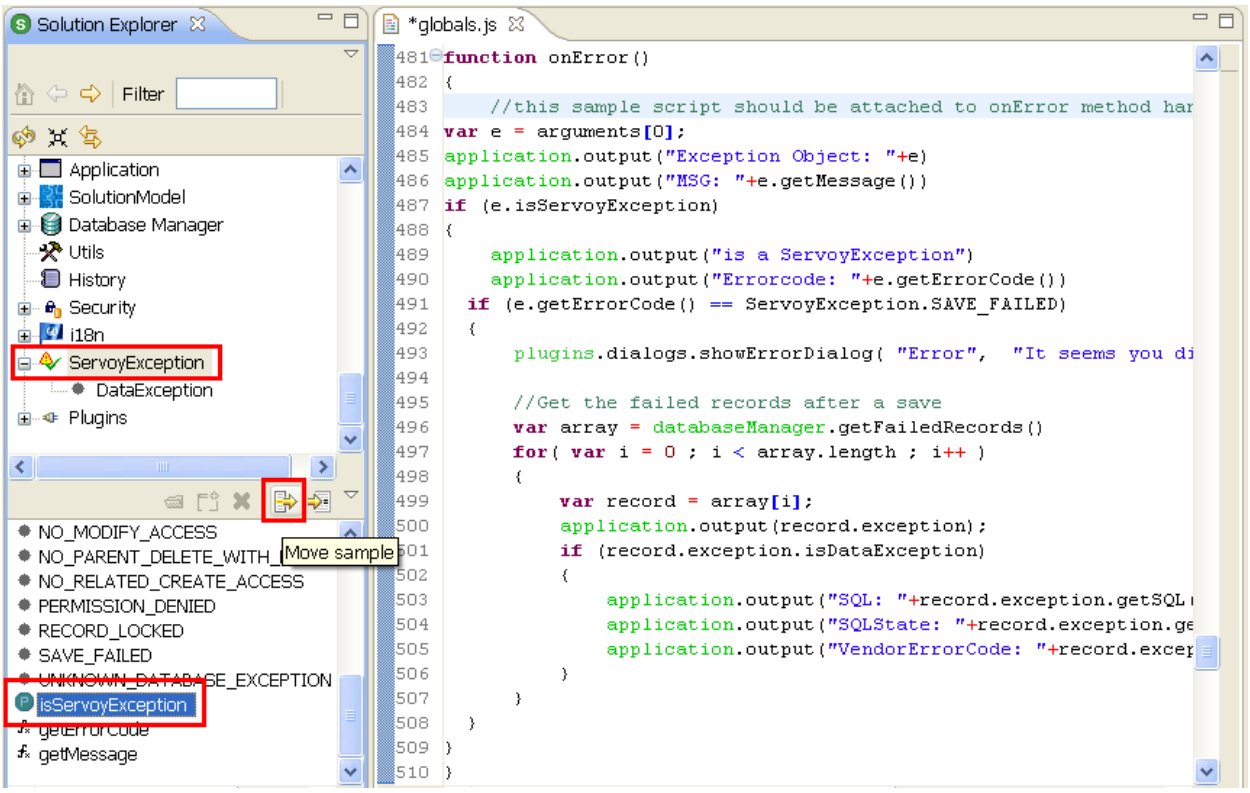
Handling Programming Errors

You can assign a method to become the error handler for any Servoy errors that occur in your running solution. To do this, you must create a global method for handling errors and assign it to the onError property of the solution. The convention is to name this mehod globals.onError, but you can call it whatever you like.



Next, you need to code your onError method to handle your error. You might want to display the error message in a dialog, with an option to quit or restart the solution. You might want to log the error in a separate log file. You might want to email the error message and some state information to a system administrator. To get started, you will need to get the nature of the error and the error message. How do you do that?

Servoy has provided some sample code to get you started. In the Solution Explorer, in the object class called ServoyException there is a property called isServoyException. Select that property. Now click inside your onError method and then click on the 'move sample' button. This moves some sample error handling code into your method which will get you started (see below).



However I have found that parts of the above sample code don't work. The part that reads:

```
if (e.getErrorCode() == ServoyException.SAVE_FAILED)
```

makes it seem like it will detect errors returned by the database but in my experience db errors don't cause this error handler to fire at all so I use a separate mechanism (described in the next section) to handle database errors.

Below is a sample error handler that builds a useful error message in a variable called errorMsg, and then tests whether we're running in Developer or not. If so, it displays the message in a dialog. If not, it sends the message to a method (not shown here) called globals.sendErrorEmail() which will email the message to a system administrator.

```
function onError()

{

var e = arguments[0]; // gets the 'error object' from Servoy
```

```
var errorMsg = "AN ERROR HAS OCCURED IN <YOUR SOLUTION NAME
HERE>\n\n"

errorMsg += "Exception Object: "+e+'\nMSG: '+e.getMessage()

if (e.isServoyException)
{
     errorMsg += "is a ServoyException, Errorcode:"+
                 e.getErrorCode()

}

//type 1 = SERVER
//type 2 = CLIENT
//type 3 = DEVELOPER
//type 4 = HEADLESS_CLIENT
//type 5 = WEB_CLIENT
//type 6 = RUNTIME



if (application.getApplicationType() == 3)  // if we're running
in Developer
{
     plugins.dialogs.showErrorDialog( 'Error',  errorMsg,  'OK')
}

else
{
     globals.sendErrorEmail(errorMsg)

}

// returning true ensures that the error still gets logged in
// the servoy_log.txt file, including the error 'stack'.
// if we returned false or nothing at all, the error would
// not be logged there.

return true;
}
```

As indicated in the comment above, returning true from your onError error handler method ensures that the error still gets logged in the servoy_log.txt file, including a detailed 'error stack'. It's a good idea to return true from your error handler so that this info is available to you in the log. Returning false or nothing at all will prevent the error from being logged. in the servoy_log.txt file.

Handling Database Errors

A Servoy error handler as described above will handle programming errors but not database errors, i.e. db operations rejected by the database (e.g. not-null constraint violations, invalid foreign key assignments, uniqueness constraint violations). Here's how you handle those.

databaseManager.saveData() attempts to save any unsaved records. If the database rejects any of those records, databaseManager.saveData() returns false. You can then find out all about the failed records and why they failed by doing databaseManager.getFailedRecords() which returns an array containing all the failed records and error information about each failure.

So what I recommend is that instead of calling databaseManager.saveData() directly each time you need to save data in your solution, create a global method called globals.mySaveData() which does two things:

- calls databaseManager.saveData()
- tests for failed records
- reports and handles any problem(s) as you see fit

The following method does all that and displays a message in a dialog telling the user that an error has occurred and that the solution will now be restarted. It also gives the user the option to view detailed info about the error before restarting. Finally, if checks if we're running in Developer, and if so it skips the restart.

```
function mySaveData()
{

	databaseManager.saveData()

	var failedRecords = databaseManager.getFailedRecords()

	if (failedRecords.length == 0) return true;

	var details = "currentcontroller: " + currentcontroller.getName() +
'\n\n'

	if (failedRecords.length > 1) details += failedRecords.length + '
FAILED RECORD(S) WERE RETURNED\n\n'

	for( var i in failedRecords )
	{
		var record = failedRecords[i];
```

```
			// application.output(record.exception);
			if (record.exception.isDataException)
			{
				if (failedRecords.length > 1) details +='FAILED RECORD '+
				   (i + 1) + ':\n\n'
				details += "Error Message: "+
				   record.exception.getMessage() + '\n'
				details += "SQL: "+record.exception.getSQL() +'\n'
				details += "Parameters: "+
				  record.exception.getParameters() +'\n'
				details += "VendorErrorCode: " +
				  record.exception.getVendorErrorCode() + '\n'


				details += '\nRECORD CONTENTS:\n'
				for (j in record)
				{
					details += j + ' = ' + record[j] + '\n'
				}
			}
			details += '\n\n'
		}

		  var msg = 'SORRY - AN ERROR OCCURED WHILE SAVING DATA.\n' +
		  'A detailed error report has been sent to the system
administrator.\n' +
		  'Dog Patrol will now be closed and re-opened. You may need to redo
your most recent changes.'


		var result =
plugins.dialogs.showErrorDialog('Error',msg,'Restart','Show Details')

	    if (result == 'Show Details') globals.bigDialog(details)  // displays
details in a large FormInDialog

		if (application.getApplicationType() != 3) // not in developer (3)
		{
			// close and re-open the solution
			application.closeSolution('dog_patrol')
		}
		else // yes in developer
		{
			globals.bigDialog(details)
			return false
		}

}
```

A couple notes about this method:

First, you may have noticed that I am not testing whether databaseManager.saveData() returned false – I am instead calling

databaseManager.getFailedRecords and testing whether the array it returns is empty or not. The reason I don't bother testing if databaseManager.saveData() returns false is because I've discovered that if there are already failed records at the time that databaseManager.saveData() is called, then it returns true! So testing if saveData() returns false is only a reliable approach if, once you've detected it, you force your user to close the solution. This is probably a good idea anyway, seeing in most cases, a db-level error is serious enough that you should probably stop the user from doing anything else. But if you do let your user stay in the solution after a db error has been detected, then it is more reliable to check the failedRecords array after each call to saveData().

Second, you will notice a call to globals.bigDialog(). This is just a function I wrote that displays a message in a big formInDialog window, and I'm using it to display detailed error info to the should he request it.

Working with Dates

by Adrian McGilly

Overview Of Dates In Servoy

This section deals with searching on dates, whether it's a user searching on a form, or a developer coding a search in a method.

Before we jump in, there are two important things you need to know about date fields in Servoy:

- By default, Servoy uses datetime fields (not date fields) to store dates.
- The display format of a date field affects how data entered into that field is interpreted for both editing and searching in that field.

Let's look at these two facts one at a time.

Servoy uses datetime fields by default

By default, when you create date fields from within Servoy you are really creating datetime fields, i.e. they will store a date component and a time component in the same field, like:

```
2005-12-20 00:00:00.000
```

or

```
2006-03-14 17:23:45.212
```

(the last three digits being milliseconds). For the rest of this discussion we'll leave off the milliseconds, seeing they don't play a role in most business applications

When you are entering data in Servoy forms and you enter a date into an empty datetime field, by default no time component will be saved. So if you enter

```
12-20-2005
```

by default Servoy will store

```
12-20-2005 00:00:00
```

However, if you happen to be entering a date into a field that already contains date-time data, and you just enter a date (no time), the time portion of the pre-existing data will be retained. So if the field previously contained

```
2006-03-14 17:23:45
```

and you enter

```
12-20-2005
```

what will actually be stored is

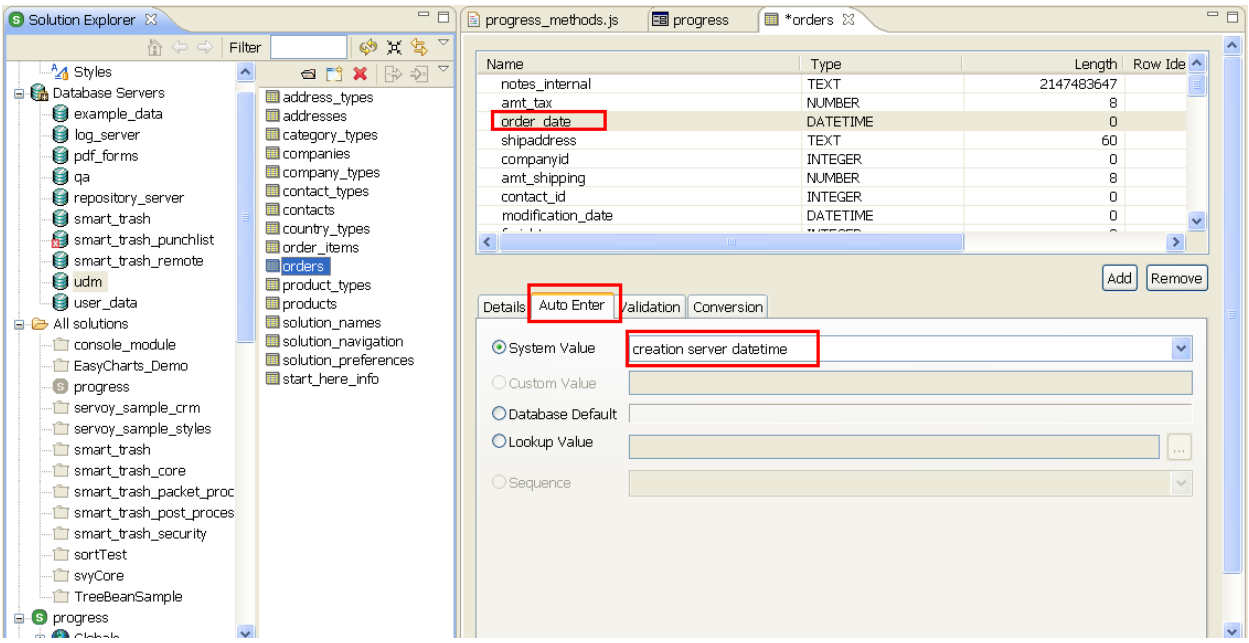
```
12-20-2005 17:23:45
```

That particular scenario is unlikely to occur in day to day use but I mention it just in case it does. The important point is this: if you want to be sure that your date-time field has a zero time component before you save it or do any local processing on it, run this line of code at some point before saving the record:

```
datefield = new Date(datefield).setHours(0,0,0,0)
```

That will zero out the hours, minutes, seconds and milliseconds of the specified `datefield`. (To better understand this line of code, read this section on editing dates programmatically)

TIP: Servoy can automatically record the time and date that any record was modified and/or created: open a table in the table editor look at the Auto-Enter tab for any column - you will see that these timestamps are offered as auto-enter values:

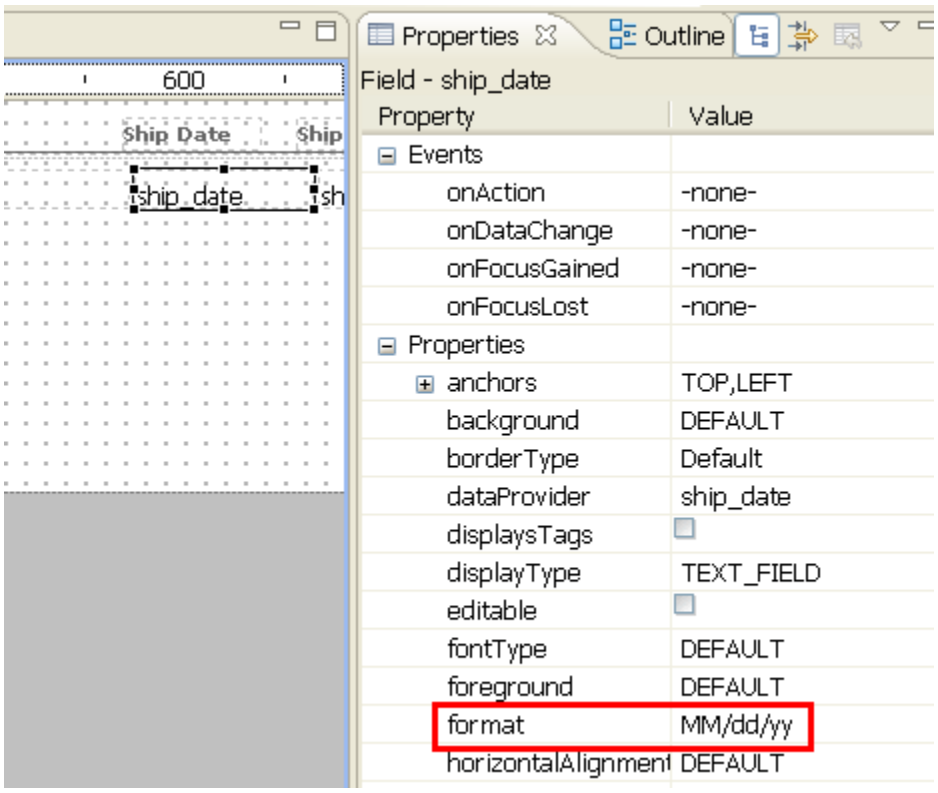


These are good examples of datetime fields which will always have a non-zero time component.

Display format affects how data entry is interpreted

Second, you need to know that the format in which dates are displayed (e.g. MM-dd-yyyy) on forms is something you, the developer, control. It is completely independent of the format in which the date is stored in the db.

You control the default display format for your entire solution with the Locale settings under Edit>>Preferences, but you can override that default format for any date field element on a form by specifying a different date format in the field's Format property on the property panel.



Note that setting the Format property will only affect how the date is displayed in that field element, on that form. If you have a database column called 'start_date' that appears on several forms, it can have a different format property on each form if necessary.

Now that you know those two important facts about Servoy date fields, we can look at how to search on dates

We need to look at two situations separately here: the situation where a user is performing a search using Servoy's built-in Find/Search commands, and the situation where you are coding a search in a method.

How Users Can Search On Dates

Servoy provides a built-in Find command available by default under the Select menu. If a user initiates a Find and types a date into a datetime field, Servoy will interpret the date using the display format of field she typed into. So, if she types

into a field whose display format is MM-dd-yyyy, it will be interpreted as Dec. 5, 2005, but if the field is formatted as dd-MM-yyyy, the entered data will be interpreted as May 12, 2005.

If she wants to be safe, she can specify the format she has in mind by using the pipe character '|' followed by the format string. So she could enter

```
12-05-2005|MM-dd-yyyy
```

and that would force Servoy to interpret her search criteria as Dec. 5, 2005.

In the above examples, seeing no time component was specified in the search criteria, Servoy will assume a 'zero' time component, and will thus return rows containing 12-05-2005 00:00:00. Any rows containing a non-zero time component will not be returned, even if they fall on 12-05-2005.

If the user wants to search on a range of dates, she can enter

```
12-05-2005…12-10-2005
```

or, to avoid any chance of confusion over the intended date format

```
12-05-2005…12-10-2005|MM-dd-yyyy
```

In this case, Servoy will return all rows containing datetime values in the range 12-05-2005 00:00:00.000 right up to 12-10-2005 23:59:59. This is good to know when searching on datetime fields with non-zero time components.

You can also put <, >, =<, >= in front of the date criteria to create open-ended search ranges. These are discussed in the Developer User Guide.

Programming A Date Search In A Method

The only thing that changes when you are coding your search in a method is that you MUST specify the date format using the pipe '|' character when your search is coded. Let's say you want to search on end_date = '12-05-2005'. If your code does this:

```
controller.find()
end_date = '12-05-2005'
controller.search()
```

your search will fail. For coded searches you MUST specify the date format using the pipe character. So the following would be correct:

```
controller.find()
end_date = '12-05-2005|MM-dd-yyyy'
controller.search()
```

Of course in reality you aren't likely to code a search on a literal date like the above examples. What is more likely is that the date (or range of dates) you are searching for are held in some containers (e.g. a global var, a local var or a db column.) So, how do you search on a date that is held in a dataprovider or var? Answer: the same way we searched on a literal date in the example above, but there's a trick you need to know.

In the above example, when we say

```
end_date = '12-05-2005|MM-dd-yyyy'
```

what we are really saying is: "here is a date that we've already formatted into MM-dd-yyyy format, and we are telling you (using the | character) what format the date is in".

Well, when the date we want to search on is held in a dataprovider or a var, it has no format. It's just a huge number in memory (corresponding to the number of milliseconds since January 1 1970 GMT).

To search on it, first we need to get it into the MM-dd-yyy format, and then we will use the | character to tell Servoy that it's in that format, just as we had to do when we were searching on 12-05-2005.

The way we do this is using the dateFormat function under the Utils note of the Solution Object Model. So if for example our date is held in a global called global.targetDate, then:

```
utils.dateFormat(globals.targetDate, 'MM-dd-yyyy')
```

returns a string which is the date formatted into MM-dd-yyy format. Knowing this, we can code our search as follows:

```
controller.find()
end_date = utils.dateFormat(globals.targetDate, 'MM-dd-yyyy') +
'|MM-dd-yyyy'
controller.search()
```

(In case you aren't familiar with string arithmetic in JavaScript, the + sign simply concatenates strings together, so in JavaScript, 'Hello' + 'World' = 'HelloWorld'). So if globals.targetDate contains Jan 23, 2006, then the above code calculates end_date as '01-23-2006|MM-dd-yyyy', which is exactly what we need. It may seem crazy to have to specify the date format twice in one line of code like that, but really it makes a lot of sense once you get your head around it.

Dates With Non-Zero Time Components

If the datetime column that is the target of your search actually DOES contain non-zero time components (e.g. in the case of a creation/modification timestamps), then there are some additional considerations because now you're no longer searching on individual dates but on a range of time that is broken down in seconds. If you ask for

```
12-05-2005|MM-dd-yyyy
```

that means 12-05-2005 00:00:00, and that's the only value that matches the criteria. A row containing 12-05-2005 11:33:44 will not be returned, even though it falls on the date you requested.

If you ask for

```
12-05-2005…12-06-2005|MM-dd-yyyy
```

that means 12-05-2005 00:00:00 to 12-06-2006 00:00:00, which leaves out the entire 24 hours of 12-06-2006! So you need to be careful.

There are a few ways around this.

1. You can tell Servoy 'I don't care what time it is, I just care about the date' by putting a '#' character at the start of your search string. So a search for

   ```
   #12-15-2005|MM-dd-yyyy
   ```

   will yield all datetime values that fall on 12-15-2005, regardless of the time component. Similarly a seach for

   ```
   #12-15-2005…12-16-2005|MM-dd-yyyy
   ```

will return all datetime values falling on 12-15-2005 thru 12-16-2005 inclusive, regardless of the time portion.

2. You can tack one extra day onto the end of your search range. If what you really want is 12-15-2005 thru 12-16-2005, you could get it by search for

```
12-05-2005…12-07-2005|MM-dd-yyyy
```

3. You can always do it the long way by search for

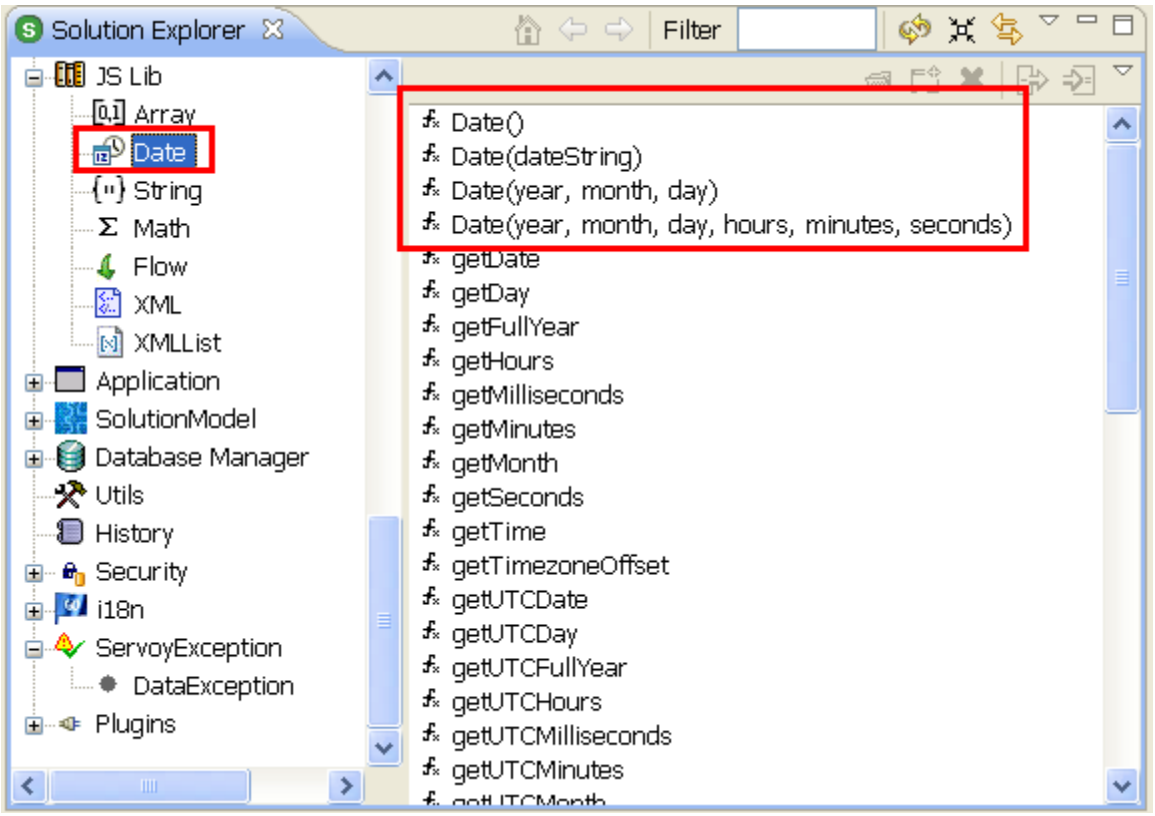
```
12-05-2005 00:00:00..12-06-2005 23:59:59|MM-dd-yyyy
HH:mm:ss
```

Tip: Use DATE Fields Instead Of DATETIME Fields

If you don't need to track times, just dates, then something to consider is changing your datetime fields into plain date fields, assuming your backend db supports them. If you do this, Servoy will continue to show them as datetime fields in the dataproviders widow, but it will treat them as date fields, and you won't ever have to tell Servoy to 'ignore the time component' because there won't be one to ignore.

Editing Dates Programmatically

Javascript provides a lot of functions for manipulating dates.



These functions act on date *objects*. The functions available for creating a new date object are at the top of the list of functions shown above. Some examples are:

```
// creates date object containing current date & time
var myDate = new Date()

// creates a date object containing Jan 15 2008.
// NOTE THAT MONTH NUMBERS ARE FROM 0 - 11, so January is 0
// and December is 11.
var myDate = new Date(2008, 0, 15)

// creates a date object containing Jan 15 2008, 14:30:25
// NOTE THAT MONTH NUMBERS ARE FROM 0 - 11, so January is 0
// and December is 11.
var myDate = new Date(2008, 0, 15, 14, 30, 25)
```

If you want to use these functions to make changes to a datetime db column, you need to convert the datetime value into a date *object*. Then you make your changes to the date object, and then save the date object back to the db column.

For example if I wanted to add one day to a datetime column called invoice_date without affecting anything else about that date (e.g. change it from Jan. 21 2007 14:30:05.000 to Jan. 22 2007 14:30:05.000), I could do it like this:

```
var dateObject = new Date(invoice_date)

// dateObject.getDate() returns the day number of the date (21)
// dateObject.setDate(x) sets the day number to x
// so setting it to dateObject.getDate() + 1 adds one day
dateObject.setDate(dateObject.getDate() + 1)

// now xfer the dateObject back to the db column and save
invoice_date = dateObject
databaseManager.saveData()
```

There's another way to modify database datetime columns, but it's not as intuitive. Dates in Javascript are stored as the number of milliseconds since midnight on Jan. 1 1970. So one way to increase the date that does NOT involve converting it to an object is to simply add 24*60*60*1000 (# milliseconds in a day) to the value and then save it back to the db:

```
invoice_date = invoice_date + 24*60*60*1000
databaseManager.saveData()
```

Make date fields 'smarter' using date,js module

Together, Servoy's calendar fields and Javascript's date functions are good enough for basic date data entry and date manipulation, but wouldn't it be nice if your users could type in things like this and have Servoy properly convert these entries into date values?

today	tomorrow
July 2008	next friday
last April	2004.08.07
6/4/2005 8:15 PM	22:30:45
+5years	

The check out the mod_datejs module put together by Greg Pierce of Agile Tortoise. It integrates a javascript library called Date.js into Servoy such that the above becomes reality. In the process you gain access to a whole bunch of intuitive date functions as shown in these examples:

```
// What date is next thrusday?
Date.today().next().thursday();

// Add 3 days to Today
Date.today().add(3).days();

// Is today Friday?
Date.today().is().friday();

// Number fun
(3).days().ago();

// 6 months from now
var n = 6;
n.months().fromNow();

// Set to 8:30 AM on the 15th day of the month
Date.today().set({ day: 15, hour: 8, minute: 30 });

// Convert text into Date
Date.parse('today');
Date.parse('t + 5 d'); // today + 5 days
Date.parse('next thursday');
Date.parse('February 20th 1973');
Date.parse('Thu, 1 July 2004 22:30:00');
```

by Adrian McGilly, Servoy Consultant & Trainer

Tabpanels Explained

by Adrian McGilly

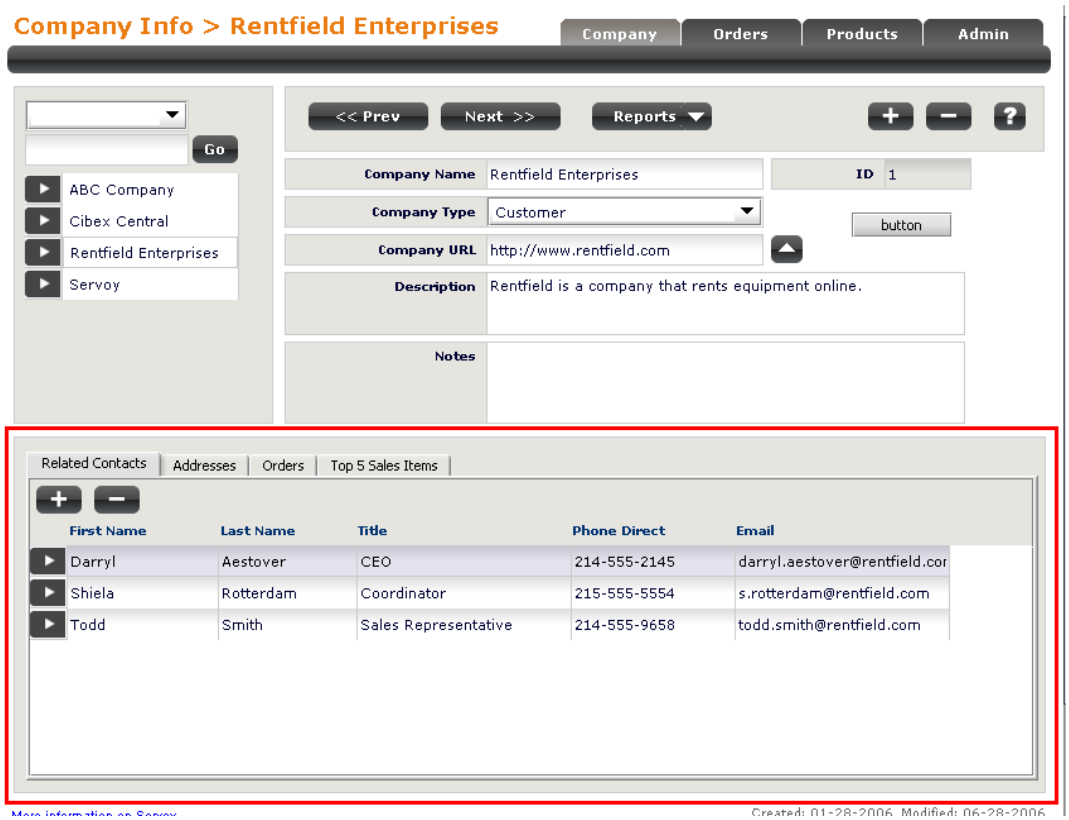
Overview Of Tabpanels

One of the most powerful features of Servoy's GUI is its tabpanels.

A tabpanel is an rectangular area on a form where you can display another form, or what some people would call a subform. Here are two ways tabpanels are commonly used:

Displaying related data

You can place several forms in the same tabpanel and let your user switch among them as needed. Below (outlined in **red**) is an example of that usage from the Servoy CRM demo solution available on Servoy's website:

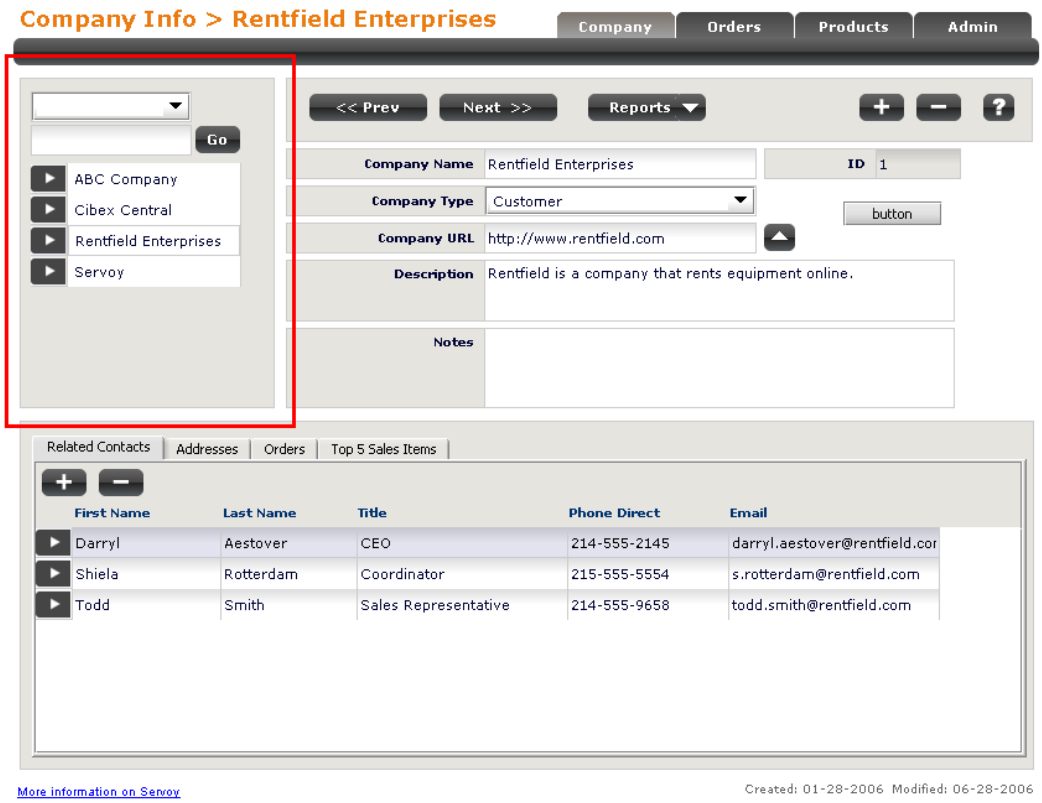


There are four forms in that tabpanel, each one accessible via its own tab, and each one displaying data that is *related* to the currently selected Company record.

Displaying Navigation Controls

You can also use a tabpanel as an area on a form where you will display navigation controls. For instance, let's say you want to put the same navigation controls on all your forms but you don't want to duplicate the controls on every form. You could create just one form that contains your navigation controls and place it in a tabpanel on each form that needs the controls displayed.

Below (outlined in **red**) is an example of a tabpanel being used to display navigation controls, also from the Servoy CRM demo solution. Notice that you can't tell just by looking at it that it's a tabpanel; for one thing, it has no tab on it, but also there's nothing about its border that reveals that it's a separate form in a tabpanel. That's because a tabpanel will only display tabs if you've placed more than one form in it. Also you can make the tabpanel blend in with the rest of the form it's on by setting its border and shading properties appropriately.



If you have placed several forms in a tabpanel, you have the option of letting the user switch among them using tabs, but you can also switch among them programatically. If you want, you can even remove the tabs, thus taking control away from the user. This is what we call a *tabless* tabpanel.

What is remarkable about tabpanels is that the forms you place in them can have lots of functionality built into them (e.g. buttons for adding, deleting or drilling down on records, validation and other business logic) and all of this functionality will continue to work inside the tabpanel. For example let's say you've created a form that contains all the bells and whistles for managing contacts. You can use it as a standalone contact management form, or you can re-use it inside a related tabpanel on your Companies form. As we will see, by doing this you will have created a subform for managing the contacts related to each company.

There are three varieties of tabpanels which we discuss in this section:

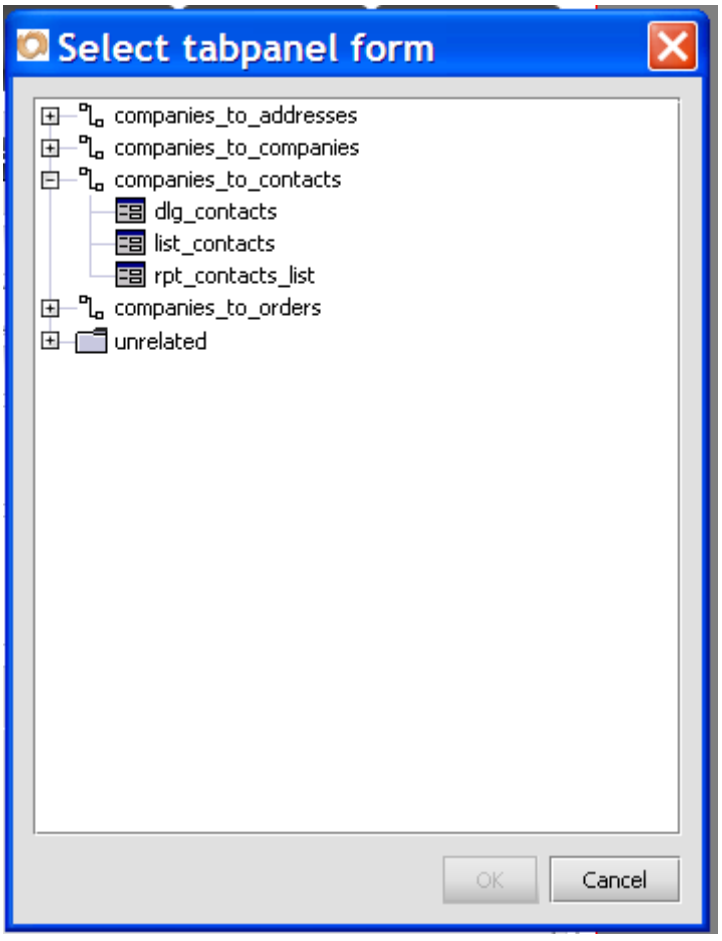
- Related Tabpanels
- Relationless Tabpanels

- Tabless Tabpanels

Related Tabpanels

Perhaps the most common use for tabpanels is the related tabpanel. When you click on the tabpanel tool to create a new tabpanel, Servoy asks you what relation you want the subform to be based on. It will only offer you relations that are based on the same table that your current form is based on.

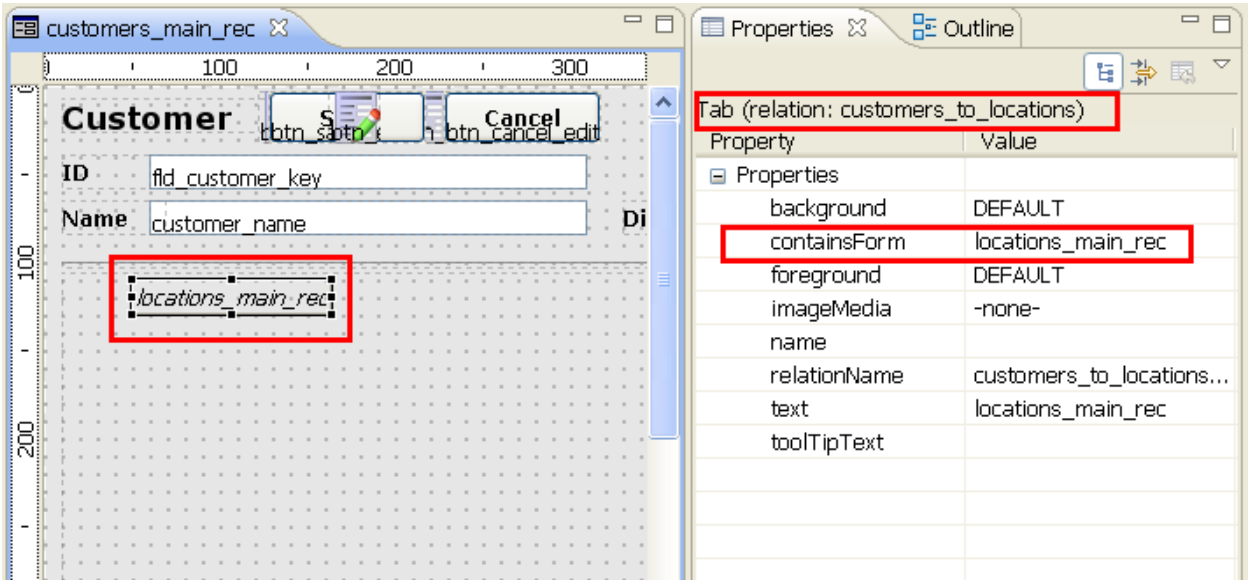
Once you click on a relation in the list, it will show you the forms that are based on the right-hand side of the relation (what I call the 'destination' of the relation in the chapter on Relations in this Handbook.)



Once you pick a form from that list, it will be added to the tabpanel, displayed as a labeled tab within the body of the tabpanel object. Below is what the 4-tab tabpanel object from the examples above (the Servoy CRM demo solution) looks like in the development environment.

Don't confuse the forms you've added to your tabpanel with the 'position example' and 'position 2' tabs at the top of the tabpanel. Those are just there to illustrate where your tabs will appear in run-time mode. Going forward, I will call the tabs that represent your forms *form-tabs*.

If you click once on one of the form-tabs (in design mode) and look at the top of the property panel (see illustration below), you will see the name lf the relation that form-tab is based on (or you will see 'relationless' if it isn't based on any relation).



If you double-click on a form-tab in a tabpanel object (in design mode), Servoy will take you to the form represented by that tab. The relation a form-tab is based on determines the foundset of records that will automatically populate the subform with when the solution is running. If the relation for the contacts subform is companies_to_contacts, then anytime that subform is visible in the tabpanel, Servoy will make sure it is displaying all the contacts records for the currently selected company.

Read this section to learn how to add and remove forms to (from) an existing tabpanel object.

Relationless Tabpanels

Sometimes the form you are placing in a tabpanel is not related to the main form, or at least not in a way that involves a Servoy relation. This can be the case for example when your subform is displaying navigation controls. In this case, when you create the tabpanel and Servoy asks you what relation the subform will be based on, double-click on the 'Unrelated' folder and you will be shown a list of tables in your solution, and beneath each table, the list of forms based on that table. (If your subform just contains a bunch of navigation buttons and no data, then it may seem odd that you're being asked to specify the form's table in order to reach the form itself in this list. Just think of it as Servoy's way of organizing your forms into a manageable list. Remember, every form is based on a table, even if it doesn't make use of that table in any meaningful way.)

Tabless Tabpanels

A student of mine once commented that the notion of a tabless tabpanel makes about as much sense as the notion of a meatless steak. Be that as it may, a tabless tabpanel is what you get when you set the tabOrientation property of a tabpanel to Hide. When you do this, you are taking control away from the user by hiding the tabs on the tabpanel, and you are taking responsibility for displaying the right subform at the right time. More on this below.

How To Switch Tabs Programmatically

Once you've added several forms to a tabpanel, your methods can control which form is displayed in the tabpanel at any given time. Just like fields and buttons you place on your form, a tabpanel is just another element on the form. One of its properties is tabIndex. If set to 1, it will display the first form. If set to 2 it will display the second, and so on. In the Servoy CRM demo solution examples illustrated above, the main form is called companies, and the tabpanel is called tabs_bottom. Here is the command to set the tabpanel to display the second form, i.e. the form corresponding to the 'Addresses' tab:

```
forms.companies.tabs_bottom.tabIndex = 2
```

You can also check what tab is currently displayed by looking at the value in:

```
forms.companies.tabs_bottom.tabIndex
```

Adding/Removing Forms To/From An Existing Tabpanel

To add another form to an existing tabpanel you need to first select the tabpanel object, then click on the tabpanel tool on the toolbar. By selecting the tabpanel object first you are telling Servoy you want to add a form to an existing tabpanel rather than create a new tabpanel object.

by Adrian McGilly, Servoy Consultant & Trainer

Relations Explained

by Adrian McGilly

Overview Of Relations

Relations are one of the most powerful and useful features of Servoy. They allow you to give recognizable name to all the data relationships in your solution, names like 'customers_to_orders' or 'orders_to_customers'. Once you've created a relationship, you can use it in many ways, but the main ways you will use it are to:

- look up related records
- create related records (with Servoy taking care of referential integrity automatically for you)
- delete related records
- count the number of related records

Primary Keys & Foreign Keys – A Quick Primer

In any relationship between two tables, you have a primary key (PK) and a foreign key (FK). Consider a customers-to-orders relationship. This is a one-to-many relationship with customer as the parent and orders as the child. Each customer has a primary key column which uniquely identifies it. Each order record must have a column in which it stores the primary key of the customer it is connected to. That column is called a foreign key. In other words, a foreign key is a pointer from a child record to its parent record.

Source & Destination: Another Way To Think Of Relations In Servoy

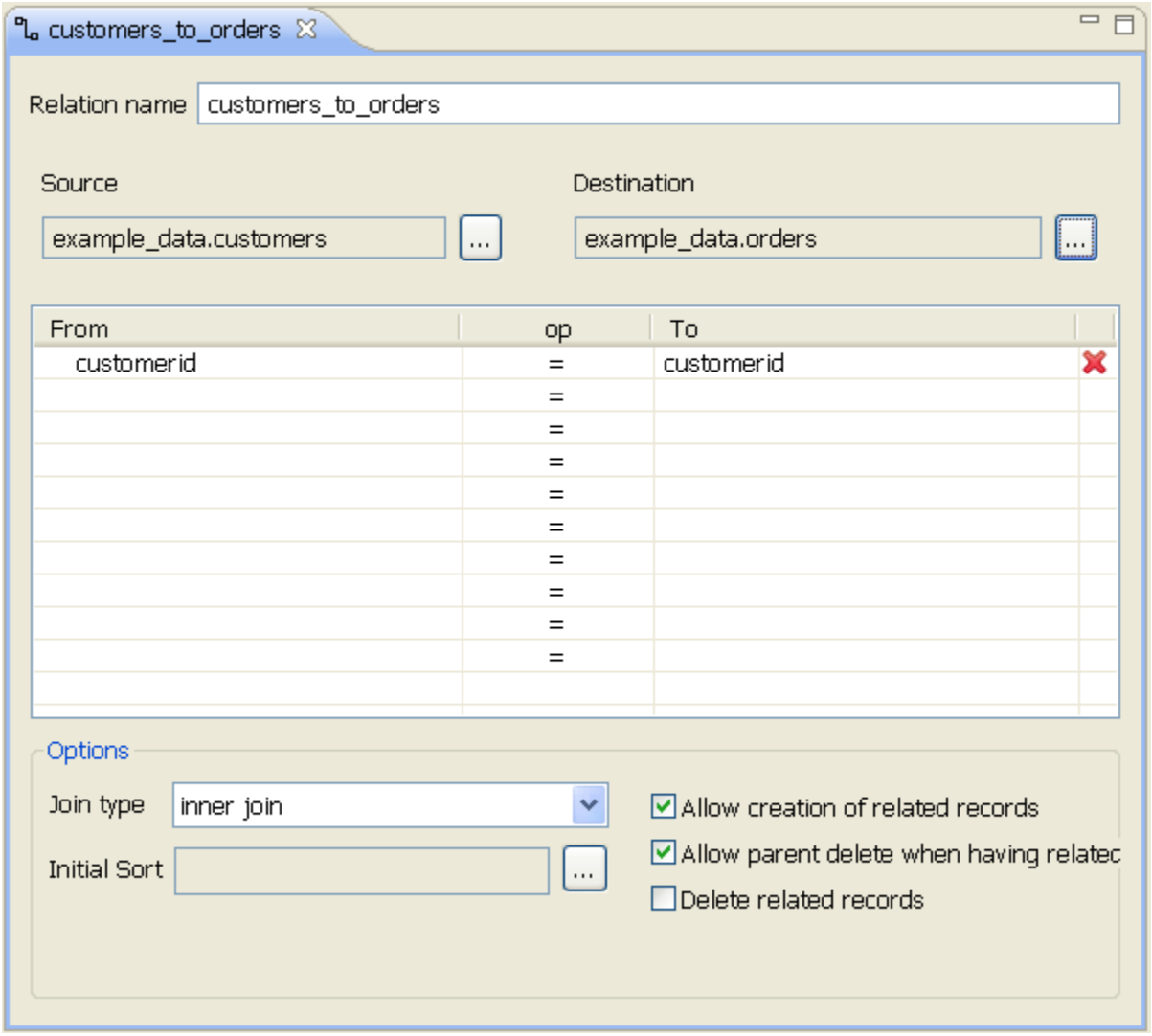
A relation can take you from a parent to its children, or it can take you from a child to its parent, depending on which way you want to go.

You'll notice that the Define Relation dialog asks for *source and destination server*, and *source and destination table* and (as of version 3.1) a *source and destination key*. If you want a relation that will lookup the orders for a given customer, you are "going from the customer to its orders". If you want a relation

that will give you the customer for a given order, you're "going from the order to its customer". Your source is "what you have" and your destination is "what you need". Source goes on the left, destination goes on the right.

One-To-Many Relations

If you want a relationship that will let you look up, add, delete, count all the orders for a given customer, you'll set it up like this:



By default Servoy will name this relation customers_to_orders based on the names of the tables involved. I usually find I don't need to change the name, but you can if you want to.

Now you're able to make use of one of the most powerful features of Servoy: the related tabpanel. You can now create a tabpanel for displaying/editing/adding orders connected to a customer. As you add new orders in this tabpanel Servoy will take care of all referential integrity.

To do this, you need to have a form (let's call it ordersForm) based on the orders table and containing any order fields you want displayed in the tabpanel (let's assume you're just dealing with order headers for now, not order items). Once you have that, you can create a new tab panel on your Customer form, base it on the customers_to_orders relation, select the ordersForm form and presto! you're done. Each time the user selects a customer on the main form, Servoy will automatically generate a foundset for the ordersForm subform containing all the order records for the current customer.

Now let's say you have placed an 'Add' button on your ordersForm whose method says

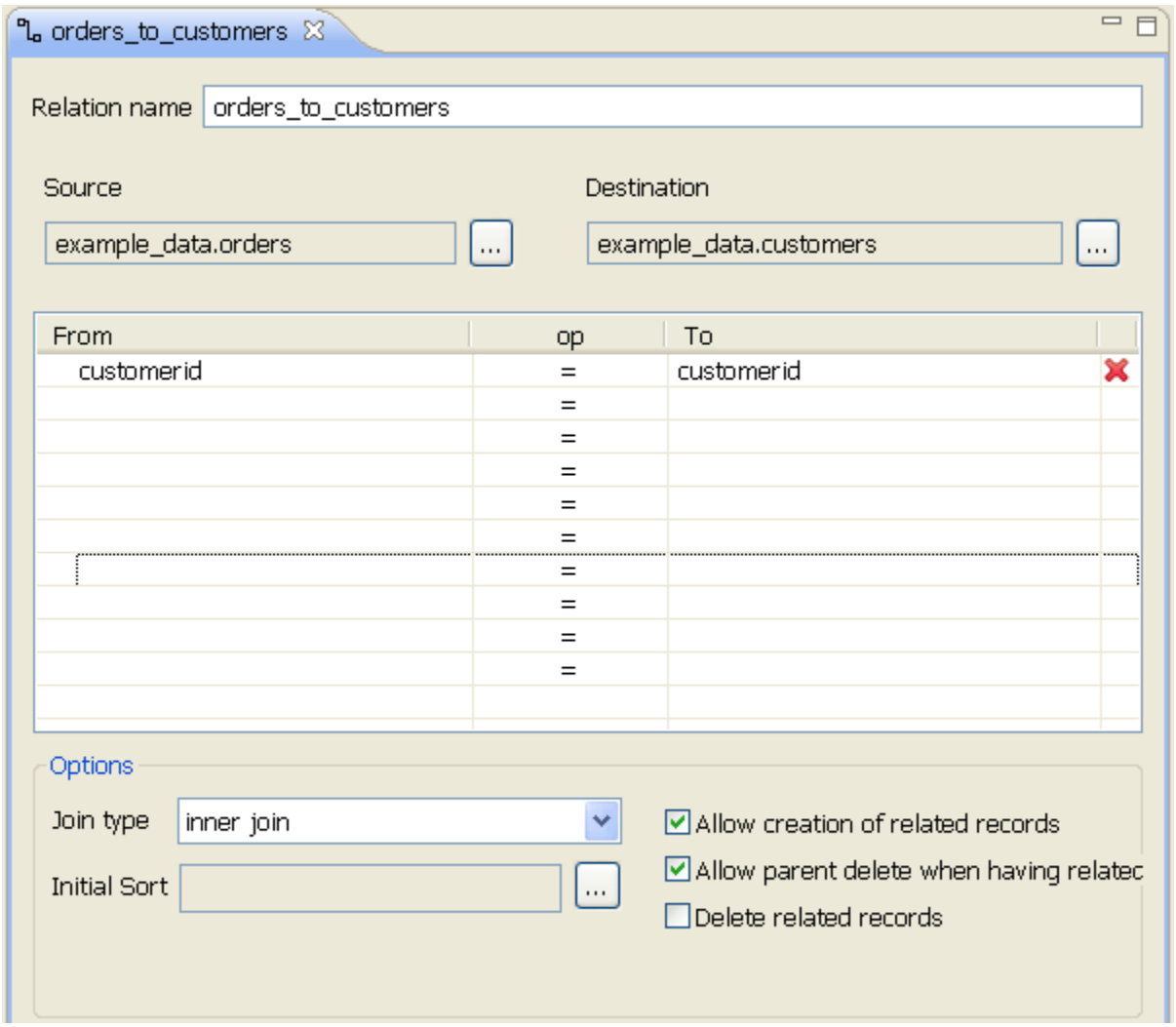
```
controller.newRecord()
```

If the user clicks on that button Servoy will know that because the form is currently being used in a customers_to_orders tabpanel, you want the new order record to point to (i.e. have as a foreign key) the PK of the current customer record, and it creates the new record accordingly. As you'll see when you read the section on Selecting, Adding & Deleting Related Records, this is the same as saying

```
customers_to_orders.newRecord().
```

Many-To-One Relations

Conversely, if you want a relationship that lets you look up the customer for a given order, you'll set that up like this:



By default Servoy will call this relation orders_to_customers. Now, if you want to display in a field the last name of the customer for the current order, you would type this in the field's dataprovider property:

```
orders_to_customers.last_name
```

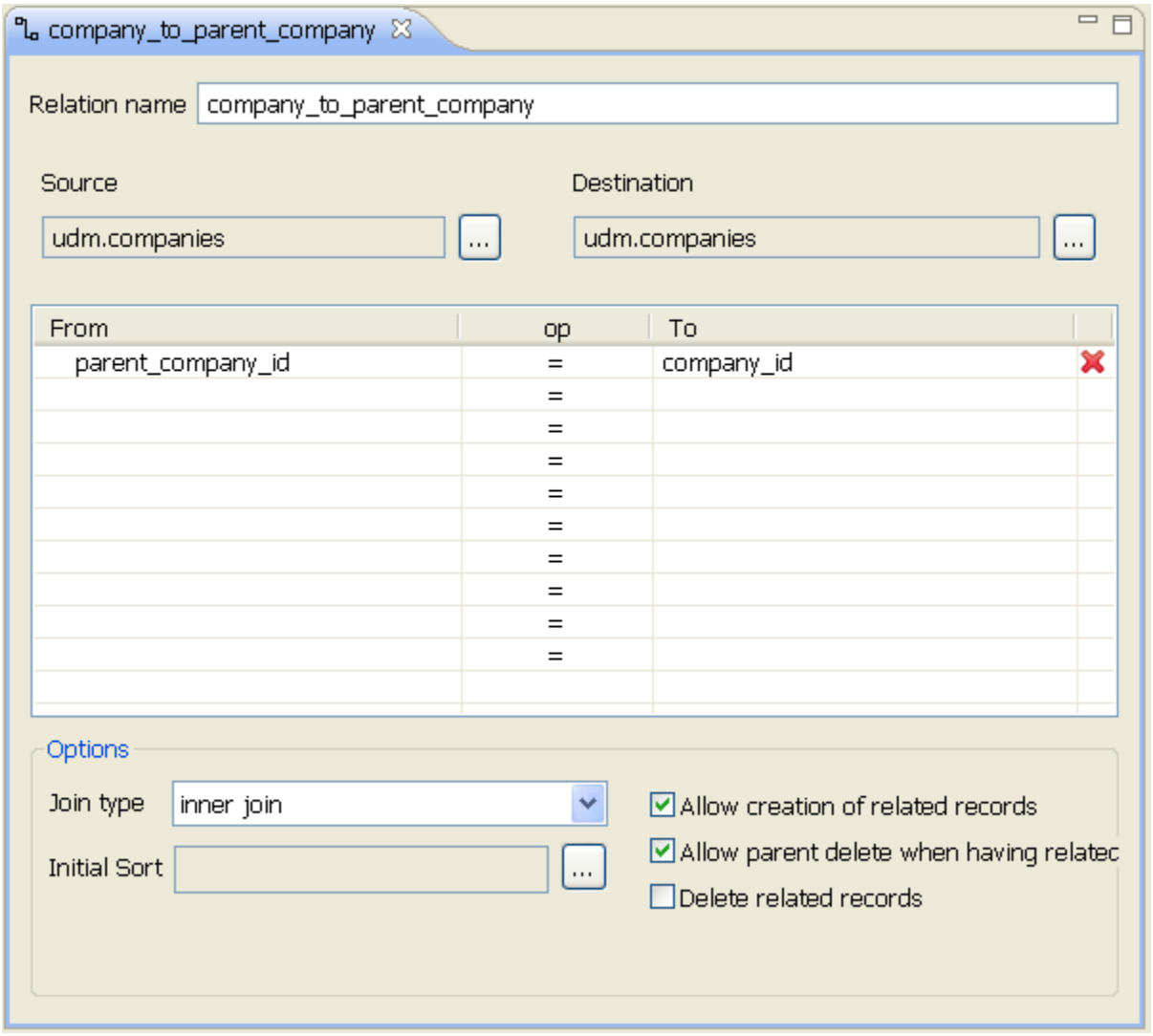
You will find that Servoy's IDE gives you access to the right relations at the right time. For instance if you are adding a new field to your Orders form (i.e. a form based on the orders table), the dialog that lets you choose the dataprovider for your field will let you select the orders_to_customers relation, thus giving you access to the related customer dataproviders.

Many To Many (N – M) Relations

It may not be immediately obvious how you would create a many-to-many relation using relations. Jan Aleman, CEO of Servoy, wrote an article on this which you can view on the Servoy Magazine:
http://www.servoymagazine.com/home/2004/10/article_nm_rela.html

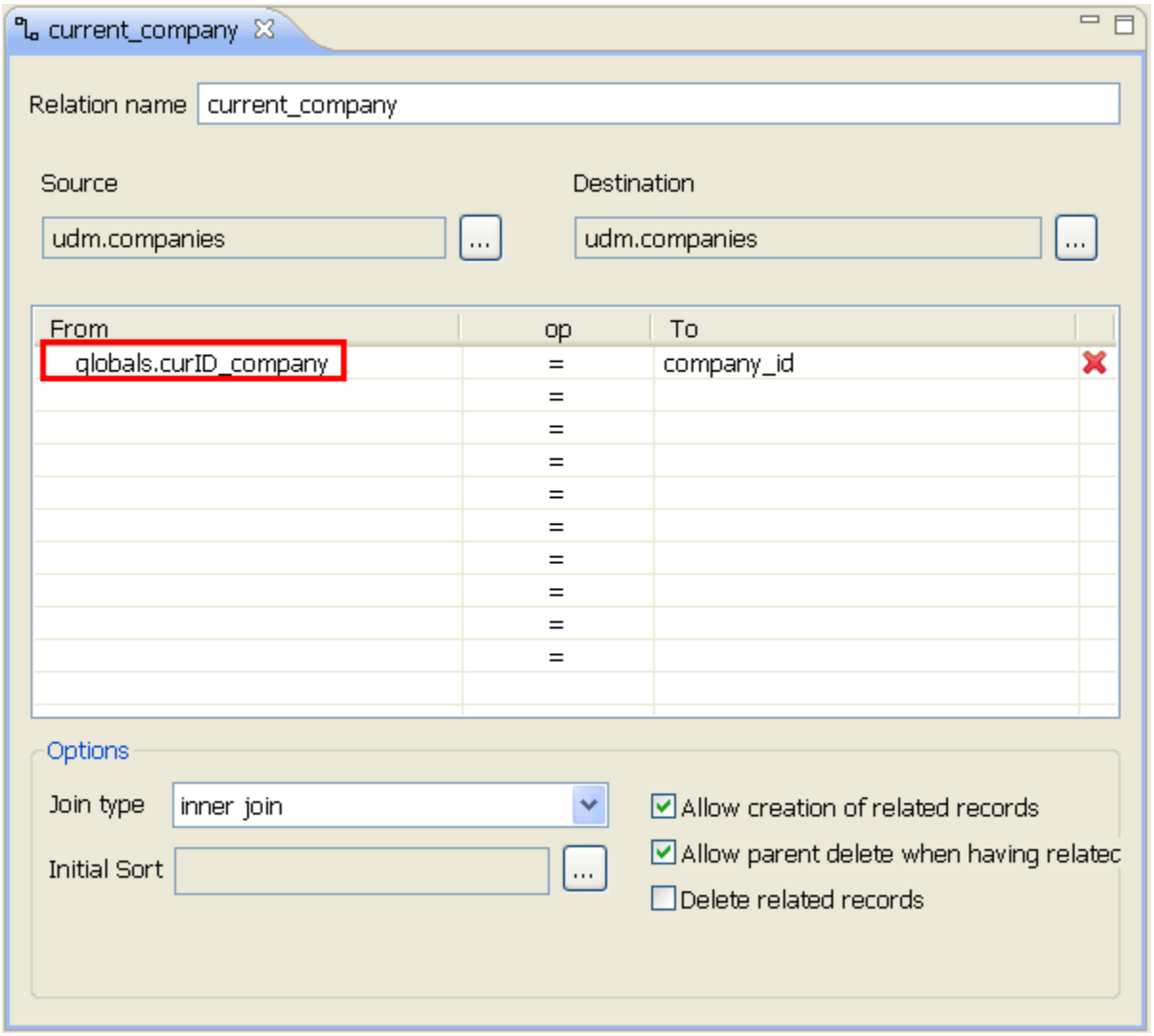
Self-Referential Relations

You can create self-referential relations, for example to reflect a hierarchical tree of employees and their supervisors. Here's a relation that will lookup a company's parent company:



Global Relations

Finally, you can create global relations, where the left hand side of the relation is not a record but a value held in a global var. This can be useful in situations where you need to access a record (and/or its relaed records), but you only have the PK of the record, you don't have the parent record itself loaded and selected in your foundset. Here's an example of a relation that gives you a 'handle' to the company record whose company_id is currently held in the global var globals.curID_company:



Via this relation, you have instant access to any company record from the moment you put its company_id in globals.curID_company. This can be very useful in a number of situations. You can use it for example to display the 'current company'

record in a tabpanel, but using the current_company relation when you create the tabpanel. You can look up the zip code in the current company's record by saying current_company.zip_code.

Nested Relations

You can chain relations together to reach a distantly related dataprovider, like this:

```
relation1.relation2.relation3.dataprovider
```

For example, if from an order_items record you wanted to refer to the customer's last name, you would do this:

```
orderitems_to_orders.orders_to_customers.last_name
```

If you were writing a method on a form based on the customer table and you wanted to set a part number in an order_items record that was two relations away to 'HB1550', you could say:

```
customer_to_orders.orders_to_orderitems.part_no = 'HB1550'
```

That's fine, but *which* orderitem record is being affected by this command? It is the 'selected orderitem record of the selected order of the selected customer'. And how do we select all these records? The best way to answer that is to look at an example.

Let's say that instead of assigning part number 'HB1550' to an orderitem, you want to loop through all the order_items records for a given customer to see if he ever ordered part # HB1550, you would do this:

```
/*set up a for loop to loop through all orders records for the
current customer */
for (var n = 1; n <= customer_to_orders.getSize(); n++)
{

   /*select the nth orders record from among the related
   foundset or orders */
   customer_to_orders.setSelectedIndex(o);

   /*set up a for loop to loop thru all orderitems for the
   selected order */
```

```
   for (var m = 1; m <=
customer_to_orders.orders_to_orderitems.getSize(); m++)

   {
   /*select the mth orderitems record from the related
   foundset of orderitems */
customer_to_orders.orders_to_orderitems.setSelectedIndex(m);

     if (customer_to_orders.orders_to_orderitems.part_no ==
'HB1550')
     {
        return true;
     }
   }
}

return false;
```

Testing if a Related Foundset Exists Before Using it

Above I described how you can chain relations together to reach a dataprovider in a 'distant' foundset, like this:

```
var myVar = relation1.relation2.relation3.dataprovider
```

There is one risk in doing this. The risk is that if relation1 is an empty foundset, or if relation1.relation2 is empty, then relation1.relation2.relation3 will not only be empty, but from Javascript's perspective it will be undefined, and that will cause a programming error when the code executes. So if there is any risk that the intervening relations in your chain might lead to an empty foundset, you should test for that condition before proceeding. You will do this using the function databaseManager.hasRecords(foundset) which takes any foundset and returns true if it contains records and false if not. It's the same as saying (foundset.getSize > 0)

So for example if there is any possibility that relation1 or relation1.relation2 could be empty, then before letting Servoy reach this line of code:

```
var myVar = relation1.relation2.relation3.dataprovider
```

You should enclose it in an if block like this

```
if (databaseManager.hasRecords(relation1) &&
    databaseManager.hasRecords(relation1.relation2))
{
```

```
    var myVar = relation1.relation2.relation3.dataprovider
}
```

You may be wondering why you can't just say

```
if (databaseManager.hasRecords(relation1.relation2))…
```

that is, why do you have to test for

```
databaseManager.hasRecords(relation1) &&
```

first? This is necessary because if relation1 is empty, then `databaseManager.hasRecords(relation1.relation2))…` will throw an 'undefined object' error. By testing relation1 first you avoid that because the way Javascript works, it 'shortcuts' ANDed boolean expressions. In other words, in any boolean expression such as

```
(a && b)
```

as soon as Javascript sees that 'a' is false it stops evaluating the expression. So in our example, if relation1 is empty, then Javascript will never evaluate

```
databaseManager.hasRecords(relation1.relation2)
```

and you will therefore avoid causing a programming error.

This need to check every relation in the chain is a bit tedious. I am told that in version 4.2 of Servoy the hasRecords() function will have a new feature where if you specify the foundset you are testing as a string enclosed in quotes, then Servoy won't generate an error if it turns out the foundset is undefined. So we will be able to say

```
if (databaseManager.hasRecords('relation1.relation2'))
```

and it will return false, even if relation1 is empty.

Selecting, Adding & Deleting Related Records

Let's look at some more examples of what you can do with relations.

These examples assume you are writing a method attached to the customer form that is based on the customer table:

```
//select the nth order record related to the selected customer
customers_to_orders.setSelectedIndex(n)

//add a new order record related to the selected customer
customers_to_orders.newRecord()

/* adds a new order_items record to the selected order related
to the selected customer */
customers_to_orders.orders_to_orderitems.newRecord()

// deletes the selected order record related to the selected
customer customers_to_orders.deleteRecord()

// tells you how many orders there are for the current customer
customers_to_orders.getSize()
```

Renaming Relations

If you decide to rename a relation, know that it will break any references to this relation in your JavaScript code. However, renaming a relation does NOT adversely affect the places where you've used the relation in the 'dataprovider' property of the property panel, (e.g. dataprovider set to orders_to_customers.last_name), nor where you've specified the relation that a tabpanel is based on - Servoy 'renames' those references for you.

Read this section for information about repairing broken methods after renaming an object.

Relations Recap

So let's recap. If you want a relation to take you from the parent to the child, (i.e. parent_to_child), then you set it up as follows in the relation dialog:

	Left side (source)	**Right side** (destination)

Table	parent table	child table
Dataprovider	parent.pk	child.fk

If you want it to take you from the child to its parent (i.e. child_to_parent), you would do this:

	Left side (source)	**Right side** (destination)
Table	child table	parent table
Dataprovider	child.fk	parent.pk

Relation Options

Three options appear at the bottom of the relation dialog (only two are shown in the illustrations above because they are taken from a version prior to 3.5)

- Allow creation of related records
- Allow parent delete when having related records
- Delete related records
- Initial Sort
- Join Type

These are explained below

Allow creation of related records

In our example of customers_to_orders, the command 'customers_to_orders.newRecord()' will create a new order record connected to the selected client record, but only if 'Allow creation' has been checked, otherwise it will give an error.

Allow parent delete when having related records

If checked, Servoy will allow users to delete records from the 'source' table even if they have related records in the 'destination' table. If unchecked, Servoy will prevent such deletions.

Delete related records

Just as Servoy will take care of referential integrity when you add related records via a relation (e.g. customers_to_orders.newRecord()), so will it ensure referential integrity when you delete a parent record. If checked, then anytime a method or a user deletes a record from the table on the left side of the relation, all related records from the table on the right will be deleted. This is a pretty powerful option - use it carefully.

Initial Sort

The order in which records will be sorted in a related foundset based on this relation. This can be particularly useful if , for instance you often need to access the most recent invoice for a given customer. Make the customers_to_orders relation sort descending by invoice_date, and you'll know that the first record in the customers_to_orders foundset will be the most recent invoice for that customer.

Join Type

This option allows you to establish the related foundset using an outer left join instead of an inner join. I have not yet found it necessary to use this in any of my development work. For details on outer left joins see this posting on the forum by Scott Butler:

by Adrian McGilly, Servoy Consultant & Trainer

Programming Tips & Gotcha's

by Adrian McGilly

Establish A Naming Convention And Stick To It

In your servoy code you are going to be referring to a whole lot of different entities by name, including:

forms (some of which will only ever appear in tabpanels, and are more like subforms than forms)

- dataproviders
- elements
- relations
- global vars
- local vars
- calcs
- global methods
- form methods

Coming up with a naming convention will help you stay organized and it will make your code more readable to you and to the next person who has to understand your code. For example, if you're not careful, you could end up with a dataprovider called last_name residing in a field element called last_name, and a local var called last_name. You'll get mighty confused if you let that happen.

Also know that all these names are case sensitive. customerForm is not the same as CustomerForm. So if you use mixed case names, you need to be consistent.

There have been some good articles written about Servoy naming conventions. You can find one which I wrote on my website, or go directly to it by clicking here. There's another one in the Servoy Magazine – here's a link:

http://www.servoymagazine.com/home/2005/04/tip_organizatio.html. I recommend you read these articles before you get too far with any solution that you intend to deploy and maintain.

If you rename any of these objects after you've referred to them in your code, you'll break the methods that refer to them. This section explains how you can repair broken methods after renaming an object.

Think Ahead And Think Big: Use Modules For Shared Objects

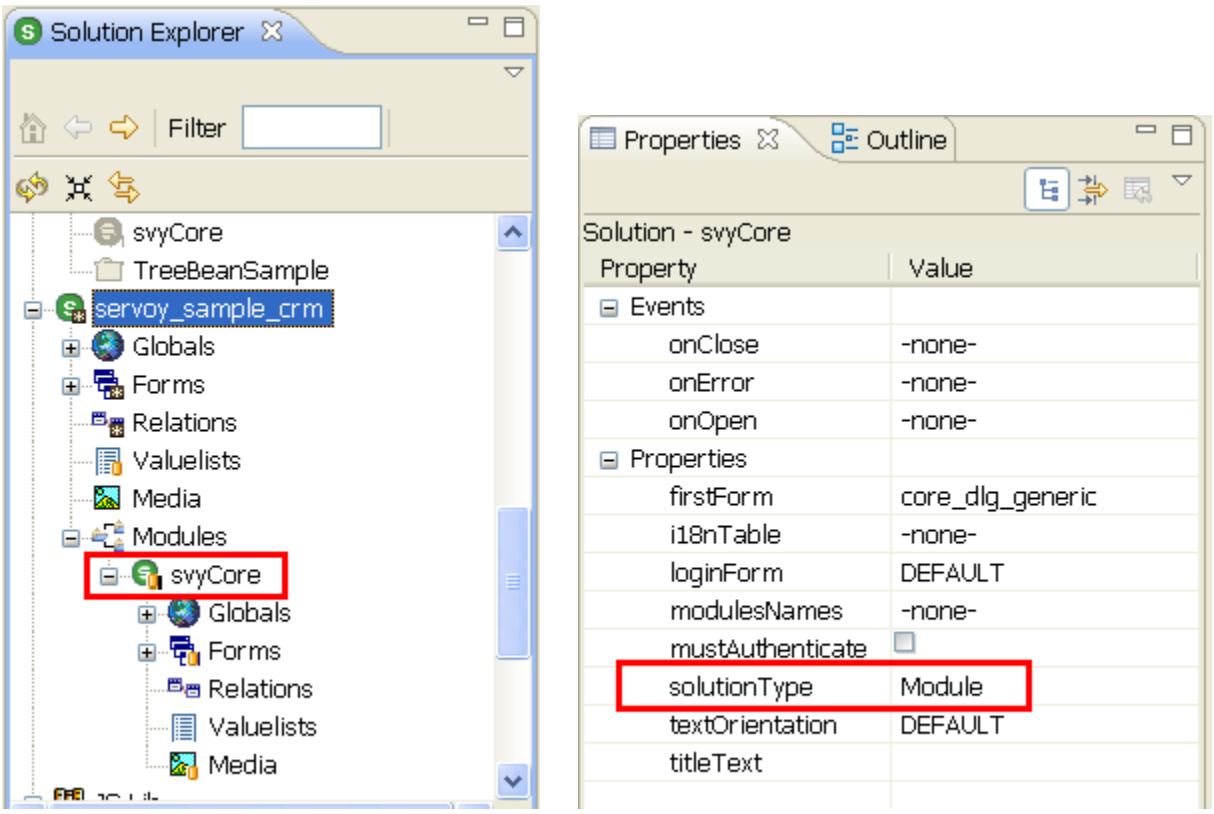
Servoy solutions can be made up of several solutions combined into one. When a developer chooses to do this, he makes one solution the "parent" solution, and he associates "child solutions" to the parent. These child solutions are called modules. Whenever the parent solution is opened, it loads any modules that have been associated to it. The master solution and its modules then act as one big solution. Parent solution A can call on any object (form, relation, method, calc, global var, valuelist, etc.) in Module B as if that object were a part of Solution A, and vice versa.

Here are some reasons you might contain part of your solution in its own module:

1. As your solution grows in complexity, so will the number of objects in it, and this can lead to some humongous lists of forms, relations, valuelists, etc. If your overall development project can be broken down into functional pieces (e.g. Accounts Receivable, Accounts Payable, Payroll) then it will likely make sense to create separate modules for these areas of functionality. This will make your project more manageable.

2. If solutions certain resources (forms, relations, methods, calcs, global vars, valuelists, etc.) are needed by several different solutions, you can create them in a module, and make them available to all solutions that need them. This keeps your code centralized and avoids duplication of resources, again making your project more manageable.

Modules are solutions just like any other, with one small difference. Before a solution can become a module, its "solutionType" property must be set to "Module", as illustrated below:



Once you've set its solutionType to 'Module', you are ready to tell Servoy what parent solution you want to add the module to. To add a module to a parent solution, select the parent solution in the Solution Explorer and double-click on its moduleNames property. You will be able to choose from a list of solutions in the repository that are of type "Module", and once added, the module will appear in the list of modules atfor that parent solution. The next time you open the parent solution, its module(s) will be opened too.

In the Solution Explorer, if the active solution has any modules, they and all of their objects will appear under the Modules node, as shown in the example above.

If you've already created a solution and you want to break it up into separate modules this is possible. First create a new solution and make it a module of your original solution. As of v 4.1 you can move forms from a parent solution to a module by right-clicking on the form and selecting 'Move Form'. In future versions this 'Move' command will be available on other types of objects as well. You could also create a duplicate of your solution in the repository, and then delete the unwanted resources from each of the two copies. To do this you'll want to export the solution and the 'clean import' it back into the same repository, giving it a

different name. After you've deleted the unwanted resources from the two solutions, make the new one a module and include it in your original solution.

Click, Don't Type!

The object tree in the Servoy Editor is very clever at writing error-free code. When you double click on a function or property or dataprovider in the tree, it moves the reference to that leaf into your code. What you may not realize at first is that it also takes into account whether the method you are moving it into is a form method or a global method, and if it's a form method, does it belong to the same form as the leaf you've clicked in the object tree… So for example, if you're working on a form method belonging to the form called customerForm, and you double-click on a dataprovider called last_name under that form's selectedRecord node, it simply inserts 'last_name' into your method. But if you do that while working in a global method (or a method belonging to a different form), it knows to put 'forms.customerForm.last_name' in your code.

For many of the functions in the library you can move a snippet of sample code into your method by clicking on the 'move sample' button in the Servoy Editor – this too can save a lot of time

When you hover your mouse over a property or function, its description and syntax appear briefly as a tooltip but they also appear in the status bar at the bottom of the Servoy window until you move your mouse away.

Use Ctrl-Spacebar To Speed Up Coding

If you do decide to type rather than click, there is still another tool in the Servoy Editor that can help you, and that is the control-spacebar command. Try typing the word 'forms.' into a method and then hit control-spacebar (command-spacebar on Mac). Servoy brings up the list of forms in your solution. Now pick a form either by clicking on one with your mouse or scrolling to it with your keyboard and hitting return. Servoy moves to the next level in the object tree and offers you all the nodes under that form's node. It will keep doing this until you reach a leaf in the tree. Pretty cool!

Be Aware Of Case-Sensitivity

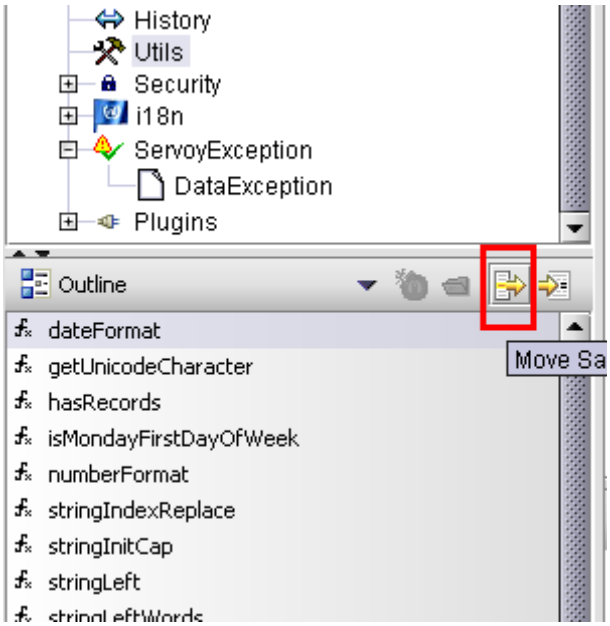
Remember that all names of all objects, functions, methods, classes, dataproviders, tables, variables etc. are case sensitive in JavaScript. A call to

forms.customerForm.Validate() is not the same as forms.CustomerForm.validate() and if you're not careful, you can spend a long time staring at JavaScript code that's generating a programming error, and not see that the problem is a simple matter of one letter being capitalized that shouldn't be, or vice versa.

This is also another good reason to click rather than type.

Use The "Move Sample Code" Button

For most properties and functions built into the Sevoy editor, Servoy provides built-in snippets of sample code which you can pull into your methods with the press of a button:



This sample code will show you how to use each command, and this will speed up your learning.

Reduce The Number Of Global Dataproviders

As your solution grows, it is easy to end up with a huge number of global dataproviders, and this can become unwieldy because there is no way to organize them into logical groups - global dataproviders are always listed in alphabetical order - and the list can get very long.

You don't actually need to create a global dataprovider every time you need a global variable. Read this article I wrote for Servoy Magazine to learn how to use standard JavasSript variable declaration syntax to reduce on the number of global dataproviders in your solutions.

Use application.output() To See 'What's Going On'

As you start playing around with writing methods, you will find it useful to peek at the values of variables, dataproviders, and evaluate expressions on the fly. While the debugger offers you some help in this area, I often find it more useful to 'print' these things to the output pane of the debugger using application.output(). Some examples:

Command	**Output** (visible in the output pane of the debugger)
application.output('Hello World')	Hello World
application.output('Hello' + 'World')	HelloWorld
application.output('last_name = ' + last_name)	last_name = Smith
application.output('qty * price = ' + qty * price)	qty * price = 270

Setting Properties At Run-Time

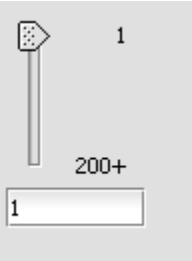
You can control many properties of your solution dynamically at runtime by setting those properties in your methods. For instance you can make fields appear, disappear, make them editable or non-editable, enable or disable entire forms, etc. This is explained in the Developer's User Guide.

Not all the properties you see in the property panel are editable at run-time. The properties you can change programmatically for any given object are listed in the Servoy Editor when you click on the object in the object tree.

When you change a property programmatically, the change only persists until the user closes the solution, and then it reverts back to its original state. In Servoy Developer mode if you leave run-time mode and go to design mode, properties set at run-time revert to their default values.

How Can I Get That Slider Control On My Custom Navigation Form?

If you're like me, you'll wish you could have that cool slider control on your custom navigation form.



Unfortunately, you can't. There's been talk of someone creating a plugin for that, but as of this writing it hasn't happened yet.

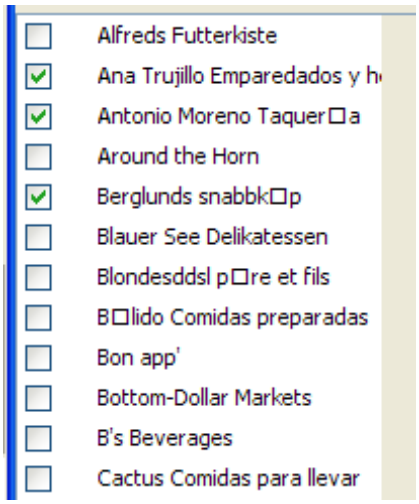
Using The %% Tags In My Labels/Buttons/Tooltips

If you are using %% tags in the text or tooltip properties of form elements, remember to check the object's "display tags" property, otherwise Servoy won't convert the tags to data values.

Use Unstored Calcs As 'Virtual Columns'

Let’s say you want to present a list of customer records to a user and let him select the ones he wants included in some process (e.g. a report). Let's assume that this is just a temporary selection, not something you want to save to the db.

From a UI standpoint, you might want to do this by putting a checkbox next to each customer name in the list, and then have the user check the ones he wants.



But what dataprovider are you going to put behind the checkbox? You could of course create a boolean column in the customer table, but that's not really what you want because you don't want to *save* these selections, you just want to use them locally for some processing and then be done with them.

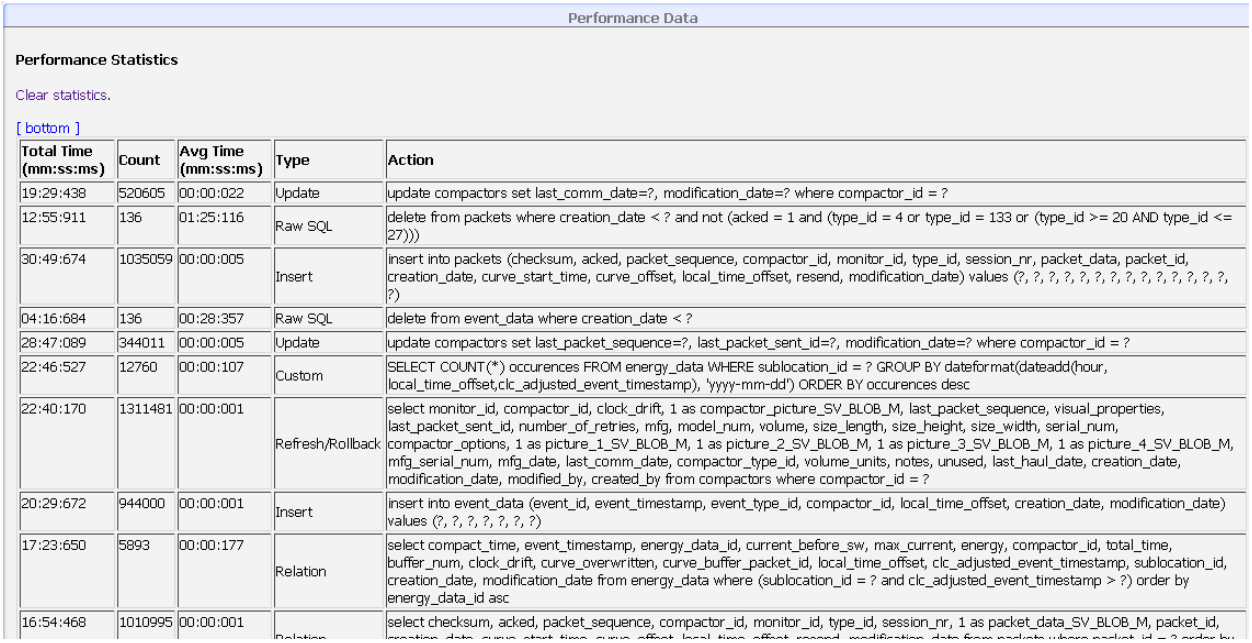
The solution is to create an unstored calc of type integer in your customer table and put nothing at all, or just a *return* command, in its script. This causes the calc to 'not calculate' but rather to accept any value that is assigned to it, be it from direct user input or from your own code.

Make this 'non-calculating calc' the dataprovider for your checkbox and you're in business. When your user checks a checkbox next to a customer in the list (i.e. foundset) of customers, the calc for that row of the foundset will be set to 1. For, unchecked rows it will be set to 0. Furthermore it will only retain its value of 1 for as long as the foundset remains in memory. If you create a new foundset, the calcs will all be reset to zero, thus ensuring that all checkboxes based on the calc will start off unchecked.

This same technique can be used anytime you want to store a value 'locally' for each record in a foundset, and of course it works for all Servoy datatypes, not just integers.

Performance Tip: Monitor Database Performance

The Admin Console of the Application Server contains a very useful page called Performance Data:



Performance Data

Performance Statistics

Clear statistics.

[bottom]

Total Time (mm:ss:ms)	Count	Avg Time (mm:ss:ms)	Type	Action
19:29:438	520605	00:00:022	Update	update compactors set last_comm_date=?, modification_date=? where compactor_id = ?
12:55:911	136	01:25:116	Raw SQL	delete from packets where creation_date < ? and not (acked = 1 and (type_id = 4 or type_id = 133 or (type_id >= 20 AND type_id <= 27)))
30:49:674	1035059	00:00:005	Insert	insert into packets (checksum, acked, packet_sequence, compactor_id, monitor_id, type_id, session_nr, packet_data, packet_id, creation_date, curve_start_time, curve_offset, local_time_offset, resend, modification_date) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
04:16:684	136	00:28:357	Raw SQL	delete from event_data where creation_date < ?
28:47:089	344011	00:00:005	Update	update compactors set last_packet_sequence=?, last_packet_sent_id=?, modification_date=? where compactor_id = ?
22:46:527	12760	00:00:107	Custom	SELECT COUNT(*) occurences FROM energy_data WHERE sublocation_id = ? GROUP BY dateformat(dateadd(hour, local_time_offset,clc_adjusted_event_timestamp), 'yyyy-mm-dd') ORDER BY occurences desc
22:40:170	1311481	00:00:001	Refresh/Rollback	select monitor_id, compactor_id, clock_drift, 1 as compactor_picture_SV_BLOB_M, last_packet_sequence, visual_properties, last_packet_sent_id, number_of_retries, mfg, model_num, volume, size_length, size_height, size_width, serial_num, compactor_options, 1 as picture_1_SV_BLOB_M, 1 as picture_2_SV_BLOB_M, 1 as picture_3_SV_BLOB_M, 1 as picture_4_SV_BLOB_M, mfg_serial_num, mfg_date, last_comm_date, compactor_type_id, volume_units, notes, unused, last_haul_date, creation_date, modification_date, modified_by, created_by from compactors where compactor_id = ?
20:29:672	944000	00:00:001	Insert	insert into event_data (event_id, event_timestamp, event_type_id, compactor_id, local_time_offset, creation_date, modification_date) values (?, ?, ?, ?, ?, ?, ?)
17:23:650	5893	00:00:177	Relation	select compact_time, event_timestamp, energy_data_id, current_before_sw, max_current, energy, compactor_id, total_time, buffer_num, clock_drift, curve_overwritten, curve_buffer_packet_id, local_time_offset, clc_adjusted_event_timestamp, sublocation_id, creation_date, modification_date from energy_data where (sublocation_id = ? and clc_adjusted_event_timestamp > ?) order by energy_data_id asc
16:54:468	1010995	00:00:001	Relation	select checksum, acked, packet_sequence, compactor_id, monitor_id, type_id, session_nr, 1 as packet_data_SV_BLOB_M, packet_id, creation_date, curve_start_time, curve_offset, local_time_offset, resend, modification_date from packets where packet_id = ? order by

This page is incredibly useful for tuning your application. It keeps track of each distinct *kind* of query that has been sent to the db from all Servoy clients (including batch processors) since the Server was last started. So let's say your deployed solution contains a form where the user selects a client and then loads all invoices for that client. That load operation generates an SQL query each time. Obviously, the exact query will differ from one time to the next depending on what client was selected, but for the purposes of generating performance stats, Servoy ignores that distinction (replacing those parameters with question marks in the SQL string) and considers all these queries as *one and the same*, and generates stats for that *kind* of query. The same applies for all SQL operations, including updates, inserts and deletes.

Each such SQL command is listed in the Performance Data page, and in the columns to the left of it are shown the following stats. These stats are based on db activity since the Servoy Server was last started:

Total Time	The total time it has taken the db to process all instances of this SQL command
Count	The total number of such commands that have been sent to the db
Average Time	The average time it has taken to process this kind of

	command
Type	The type of command. Possible types include: • Find (generated by a find()/search() pair of commands • Relation (generated by accessing a related foundset) • Aggregate SQL (generated by an aggregation) • Raw SQL (sent using the rawSQL plugin) • Custom (sent using loadRecords(mySQLString)) • Update (generated by a saveData() command following changes made to one or more records) • Insert (generated by a saveData() command following a newRecord() command) • Delete (generated by a deleteRecord() or deleteAllRecords command) • Refresh/Rollback (presumably generated by a databaseManager.rollBackTransaction() command but I sometimes see very high counts for these and I'm not sure where they come from)

Using these statistics you can quickly find out what operations are costing you the most in terms of db activity. Look first for the highest Average Times, and if any of them also have high counts, then you know you have a bottleneck.

Once you've identified some sluggish queries, figuring out exactly where they are coming from can be a bit tricky. For starters, the Performance Data doesn't tell you which exact solution is the source, let alone which method of that solution. The Type column obviously gives you a hint, and the query itself should also give you some guidance. But if you still can't pinpoint the exact source of the query you can narrow down the possibilities by trial and error, and cou can do this either in Developer or in a client hitting the Server. Go to the Performance Data page of the Admin Console, clear the statistics that are there, then perform an operation in your solution that you suspect is the source of the query. Then refresh the Performance Data page - there you will see exactly those SQL commands your code just generated.

Performance Tip: Make Judicious Use of Aggregations

Servoy supports Aggregates on tables (sum, average, count, max, min). These are very handy, but they come at a price. Every time you create a foundset based on that table, Servoy sends a separate query to the db to resolve all aggregations on that table, even if you never use the resulting aggregation in your code or on any of your forms. This is why in the Performance Data page of the Admin Console you will often see queries starting with 'SELECT COUNT(*)' or 'SELECT AVERAGE(column)' etc.

If you see from the Performance Data that a little-used aggregation is eating up a lot of db processing time, you may want to consider a more efficient approach, such as writing your own SQL command to retrieve the aggregation, and using databaseManager.getDataSetByQuery() to retrieve the result.

Be aware that aggregations, like calculations, are solution-specific. If an aggregation is defined in one solution but not in another, it will only be performed when the solution containing the aggregation is running.

If you have broken your solution up into modules, and if you have a 'core' module that is used by several different parent solutions, be careful where you define your aggregations. If an aggregation is defined in solution M, and solution M is a module within solutions X, Y and Z, then the aggregation will be executed every time the relevant table is accessed from X, Y or Z. If the aggregation is not needed by solutions Y and Z, only by solution X, then you should define the aggregation in solution X.

Performance Tip: When NOT To Use Related Foundsets

I once worked on a Servoy system that processed a large volume of incoming and outgoing transactions, each requiring some fairly intense database processing. These transactions were being handled by a batch processor running directly on the server. Given the large volume of transactions, I needed to squeeze every drop of performance I could out of Servoy and the database.

I learned that when adding new records into a table, sometimes the more elegant, readable code carries a performance hit and you are better off resorting to code that is more efficient, though less elegant. Here's an example.

Let's say I have tables A and B, where B is the child of A (i.e. A to B is one-to-many).
I have a foundset called fsA on table A.
I have a relation called A_to_B.
I have a record selected in fsA and all I want to do is add a new child B record.

To insert the new child record I could do this:

```
fsA.A_to_B.newRecord()
databaseManager.saveData()
```

This is convenient because it automatically assigns the foreign key(s) in my new B record, and it's also nice, compact, easy-to-understand code.

The problem is that it does more than I really need it to. In addition to inserting the new record it does the following:

1. it builds the A_to_B related foundset, (i.e. it performs a select for all B records related to the current A record.)
2. it resolves any aggregates for the A_to_B foundset (i.e. if I have any aggregations defined on table B, then it performs the required SQL to derive those aggregations for the A_to_B foundset)

In those situations I get better performance if I do this:

```
fsB = databaseManager.getFoundset('server_name','B')

fsB.newRecord()
fsB.foreign_key = foundset.pk_id
databaseManager.saveData()
```

This does one and only one thing: it performs the insert.

So in general, when performance is a top priority, you want to be careful about making references to related foundsets in your code. Only refer to the related foundset if you really need the related records. Otherwise, find another way to code your process.

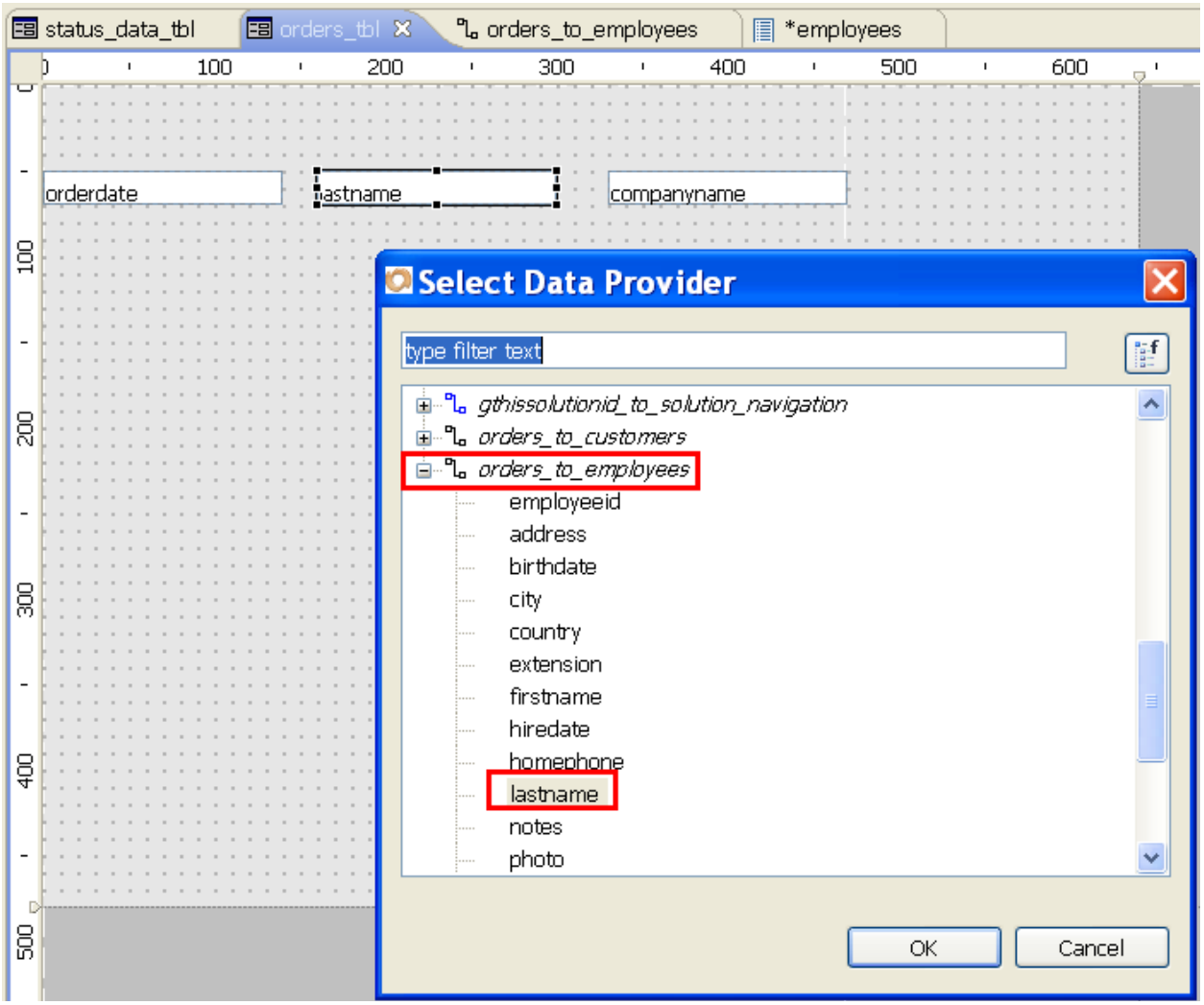
Speed Up Related-Data Lookups By Using Valuelists

If you are displaying a list of records that includes related data, you are making the database engine do some work joining tables together to get you your data. That's

fine - that's what databases are for - but in some cases you will get better performance if you use this little trick.

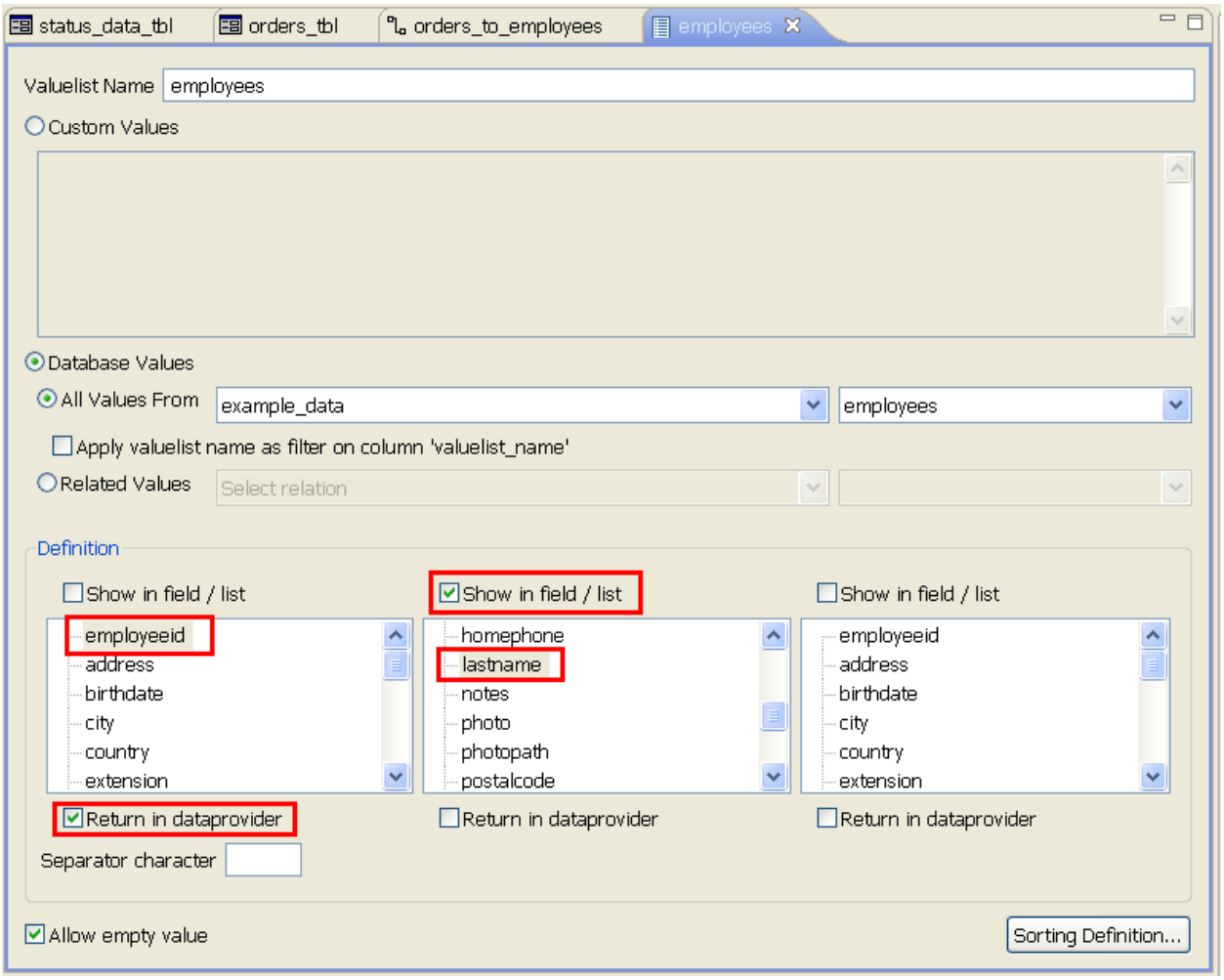
Let's say you have a table called orders, and a table called employees, and each order is assigned one employee. So you have a column called order.employee_id that is the foreign key in the orders table, and you have a column called employee.id which is the primary key in the employee table.

Now imagine that you are displaying a list of orders in a list view or a table view and one of the columns is the employee's name. You could of course get that name to appear in the list by creating an orders_to_employees relation and then placing the related dataprovider orders_to_employees.employee_name on the form, as shown below:

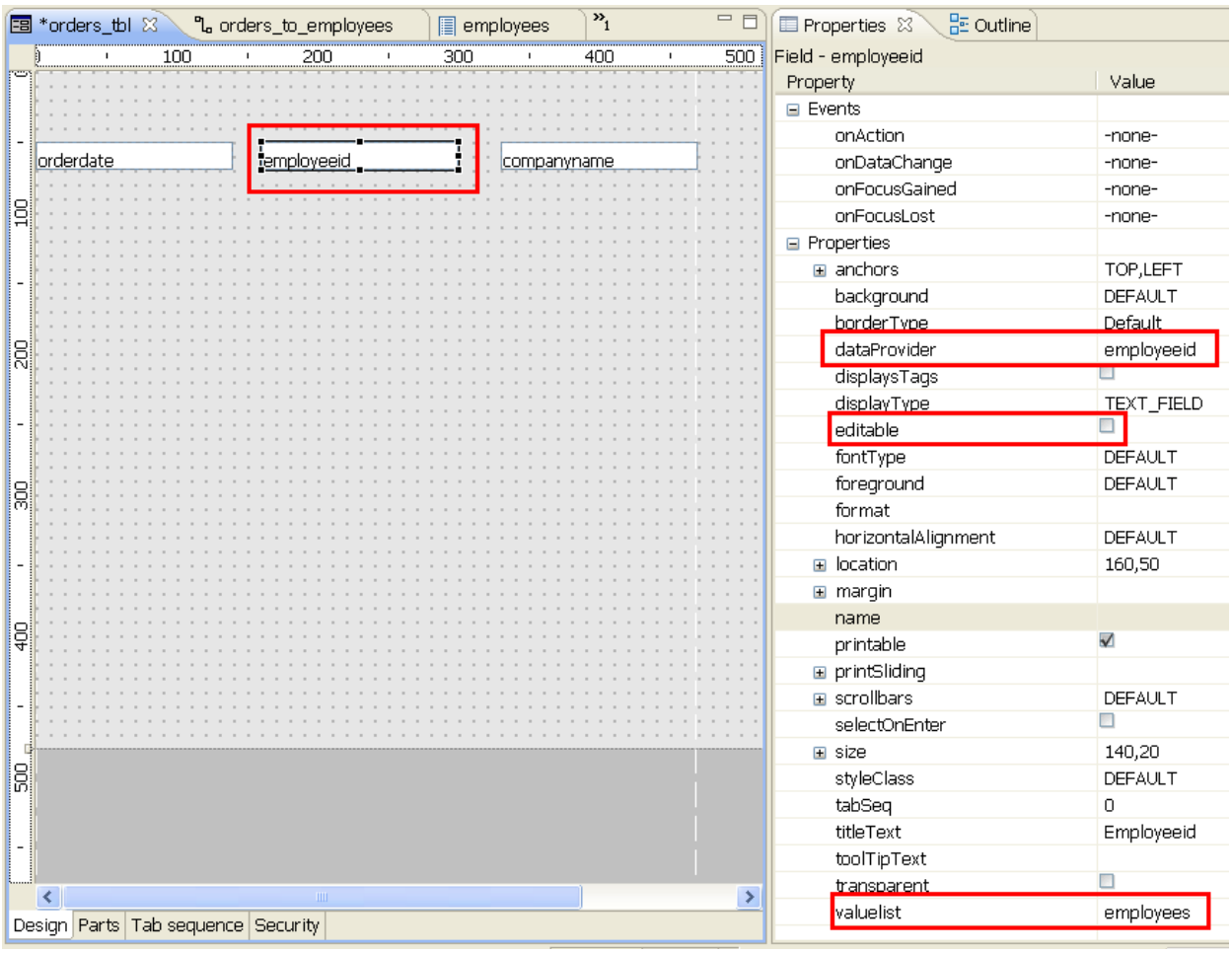


But you will get faster performance if you use the foreign key orders.employee_id as the dataprovider for the field, and use a valuelist to lookup and display the actual name of the employee. Simply create a valuelist like the one shown below that:

- is based on the employee table
- 'returns to dataprovider' the employee_id
- 'shows in list' the employee name



Now assign this to the valuelist property of the orders.employee_id field in your list view or table view, as shown below. Servoy will lookup and display the employee's name and it will do it faster than if you relied on a relation.



How To Make A Button That Is A Transparent Icon

If you want a button that is just an icon, with no border, you need to uncheck the showClick property of the button.

How Do I convert Numbers And Dates Into Formatted Strings?

Sometimes you need to construct strings in JavaScript that contain numeric data or dates, and you need the ability to format the numbers and dates in a particular way. See utils.numberFormat() and utils.dateFormat().

Removing The Time Component Of A Datetime Value

Servoy programmers often have to explicitly zero-out the time component of date-time fields, and often they do it like this:

```
var dateObject = new Date(invoice_date)
dateObject.setHours(0)
dateObject.setMinutes(0)
dateObject.setSeconds(0)
invoice_date = dateObject
```

but you can replace three lines of code with one by using the setHours() function like this:

```
var dateObject = new Date(invoice_date)
dateObject.setHours(0,0,0,0)
invoice_date = dateObject
```

Global Vars Don't Persist When You Leave Run-Time Mode

In Servoy Developer, if you open and close your solution while debugging your code, remember that none of your variables, not even global vars, persist once you leave run-time mode. They all revert to their default values which, unless you specify otherwise, is null.

Back Up Your Work Often

Remember, all your hard design and programming work is stored in the repository, a database usually called servoy_repository.db. If the repository somehow gets corrupted or deleted, you've lost all the solutions held in that repository. Although SQL Anywhere is very stable, don't take a chance on database corruption destroying days of work. Make frequent backups of your repository, and if possible, your entire database folder.

Use The Built-In Help

The built-in help under the help menu is the same document as the PDF and paper versions of the reference guide – very handy. Use it!

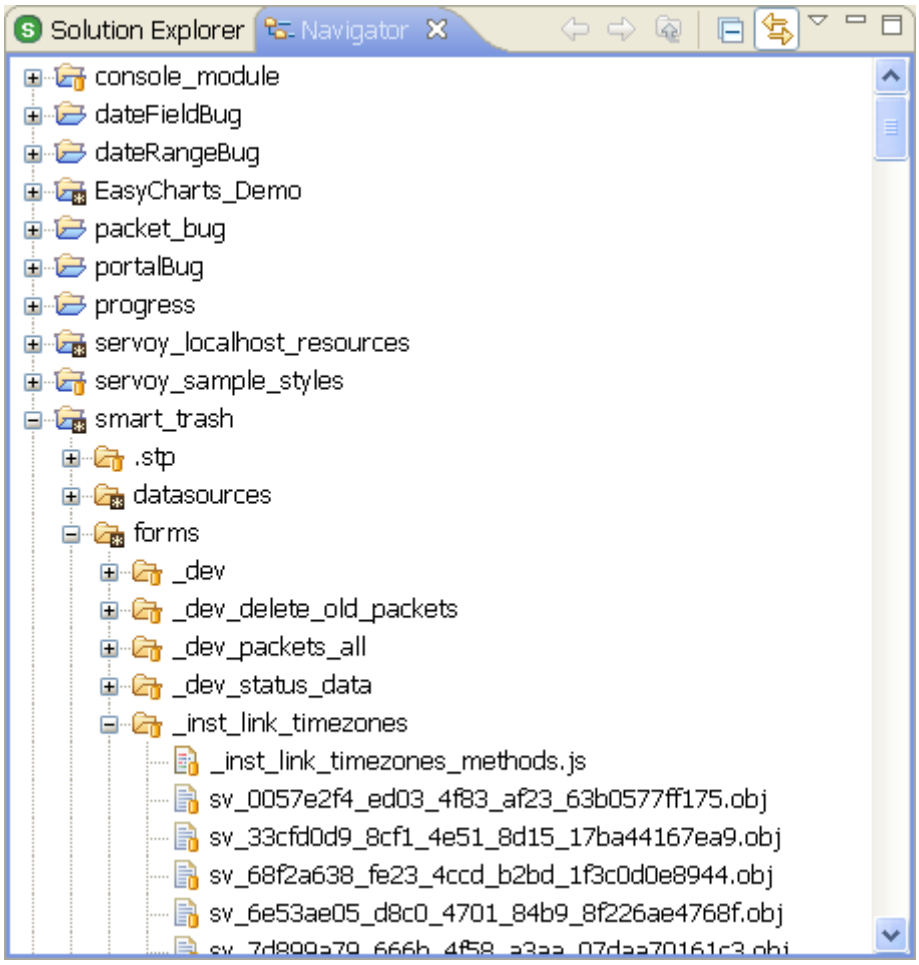
Use The Eclipse Navigator View

When developing in Servoy, you generally work on one parent solution at a time. If the parent has modules, then you will have access to them under the parent solution's 'modules' node. But if you want to work on a different solution altogether, you need to activate that solution, and that de-activates the first one. Sometimes all you want to do is take a quick look (or even make a quick change)

to a method or form in the other solution, and you'd rather not have to de-activate your current solution. You can actually achieve this by using what is called the 'Navigator' view in Eclipse.

To add the Navigator view to the curent Eclipse perspective just choose Window>>Show View>>Other>>General>>Navigator

This will add the Navigator pane to one of the panels in your Eclipse Perspective (you'll probably find it in the same panel as your Solution Explorer) so that it will look like this:



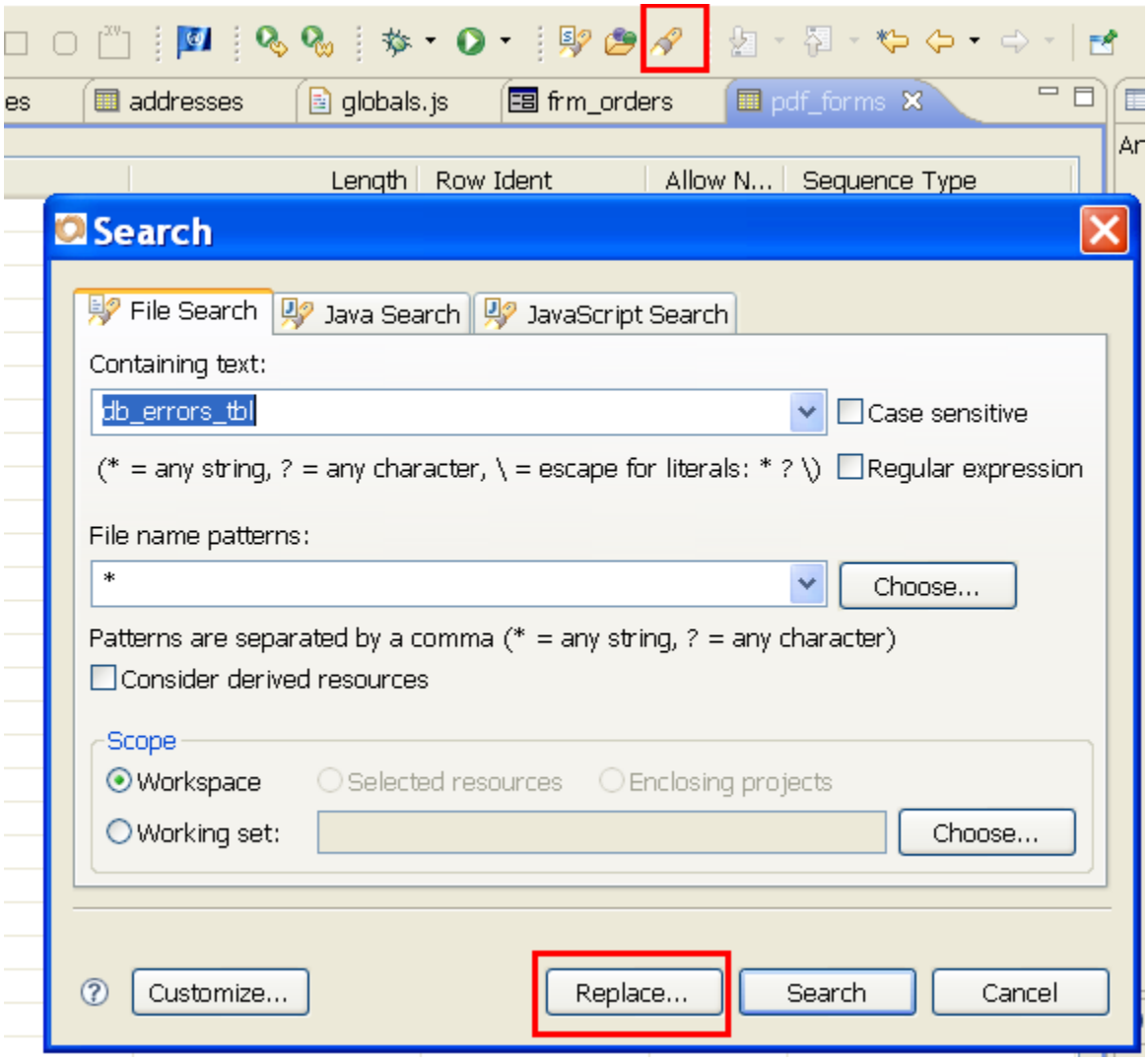
This Navigator pane gives you access to all objects and methods in all solutions currently found in your workspace (i.e. any solutions you have checked out of the repository and not since deleted from your workspace). Items ending in .obj are objects such as forms, fields, buttons, etc. Items ending in .js are methods. To

open any object or method just double click on its node in the tree. You can edit objects and methods just like you would normally.

Though not quite as user friendly as the Solution Explorer, this is a handy tool for jumping around quickly among solutions.

Use The Global Find & Replace

If you change the name of an object that is referred to in your methods (e.g. a form, a relation, a dataprovider, etc.) you will break those methods. You can use the global find & replace command found under Search>>Search (Control-H) in the Servoy IDE to search for references to the old name and replace them with the new name.



Don’t Forget The Double == In If Statements!

Remember that in JavaScript, to test equality you need to use two equal signs.

Don’t do this:

```
if (i = j) /* assigns the value of j to i and returns the value
of i to your if test */
{…
}
```

when you mean this:

```
if (i == j)  // uses two equal signs to test if i equals j
{…
}
```

This takes a while to get used to, and it can be a hard error to spot in your code, so take note!

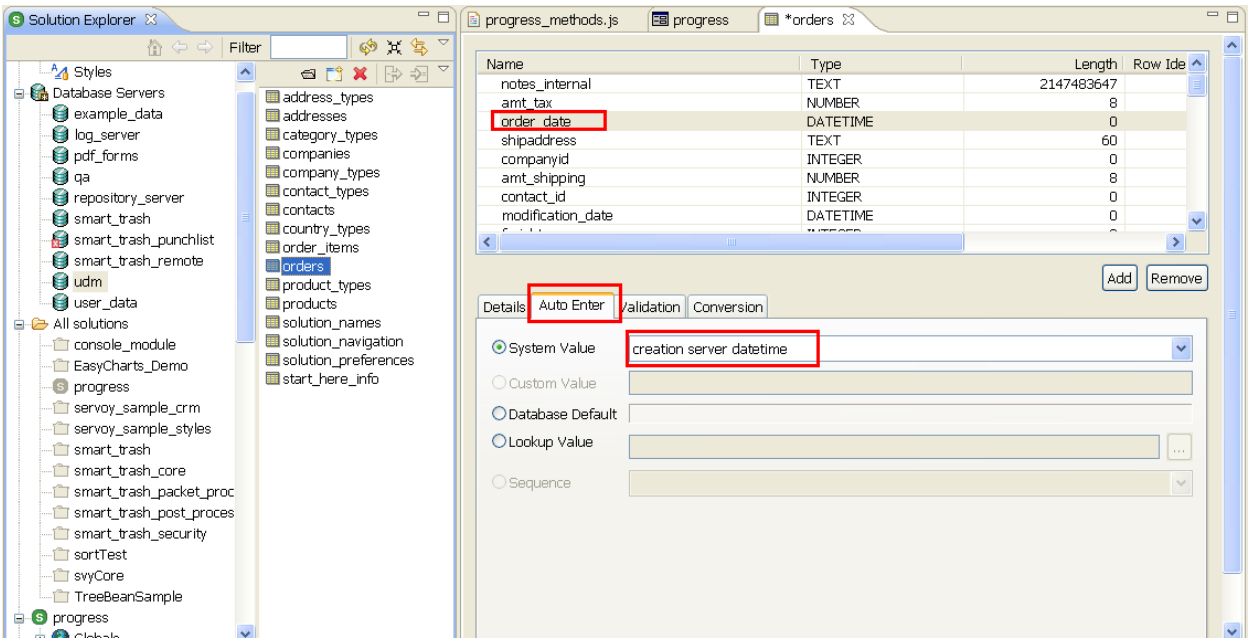
Select Code & Hit Tab To Indent It All At Once

If you want to indent a whole chunk of code to the right in a method, select the code you want to indent and hit tab – Servoy will insert a tab before each selected line.

Unfortunately there’s no way to ask the editor to “format’ your code by indenting it consistently for you.

Let Servoy Generate Timestamps

Need to record the creation/modification timestamp for certain tables? Servoy can do that for you automatically. Create two columns for the creation and modification timestamps and then look at the properties of each column by selecting the column in the table editor and clicking on the Auto Enter tab. You will see that two of the auto-enter properties are Creation Timestamp and Modification Timestamp. Select the appropriate ones and you're done!



by Adrian McGilly, Servoy Consultant & Trainer

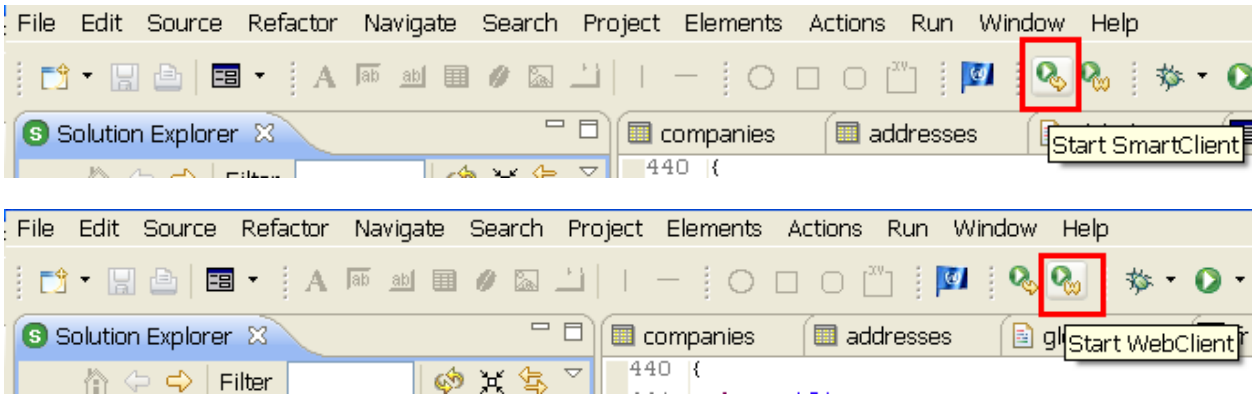
Debugger Tips and Gotcha's

by Adrian McGilly

Servoy comes with a great debugger. It is well documented in the Developer User's Guide, but here are a couple tips to help you get started:

Solution must be "running" for debugger to work

This may seem obvious, but I'll say it anyway. You won't be able to run or step through a method unless your solution is 'running'.



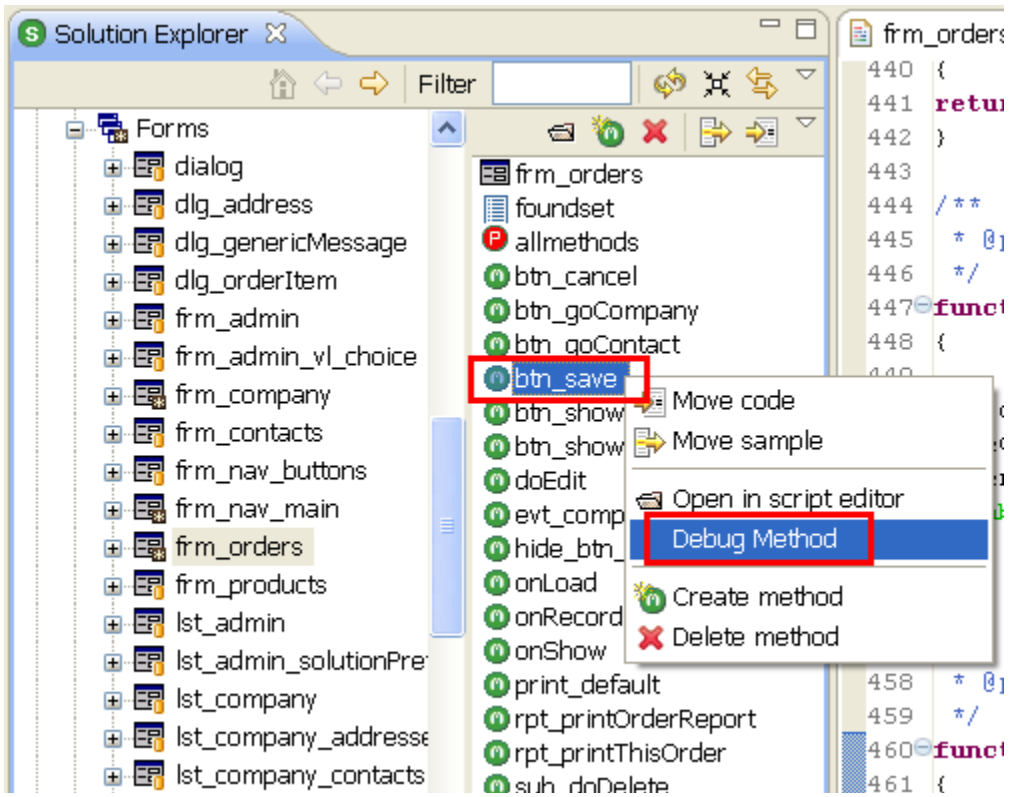
Editing methods while debugging

You can edit methods while your solution is running, however if you are have placed a breakpoint in a method and execution has suspended in that method, changes made to that method won't take effect until you save them and run that method again.

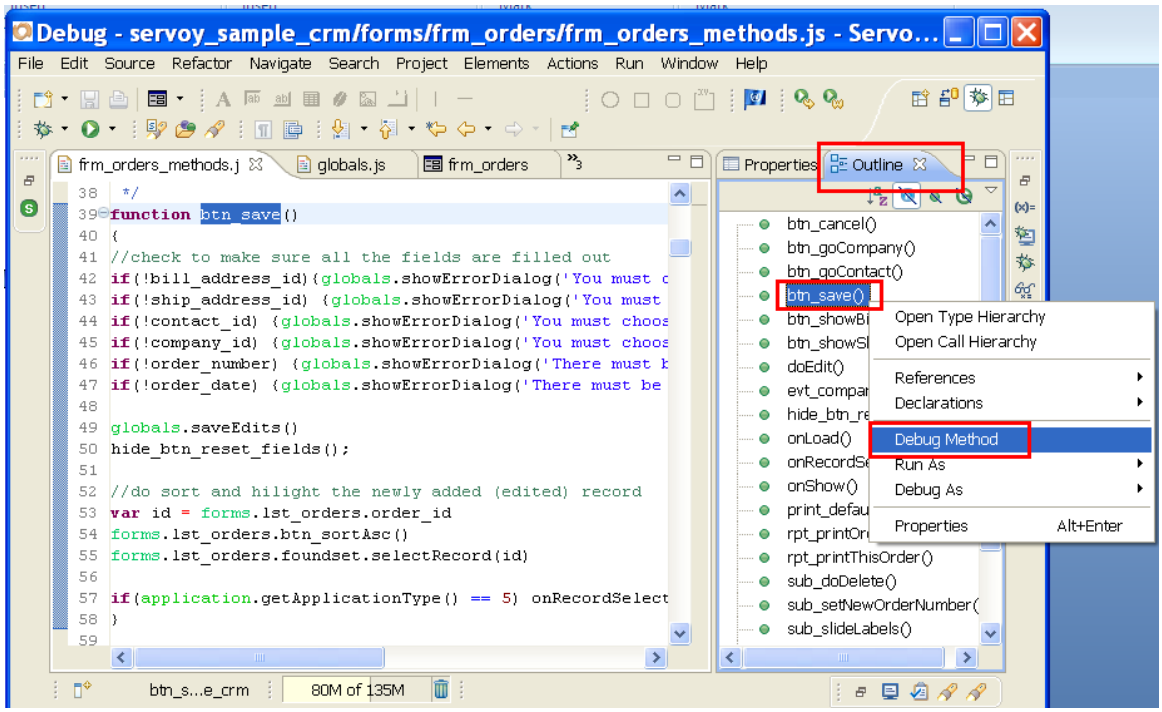
You can run a method straight from the debugger

As long as your solution is running in Developer, you can execute any method anytime. There are two ways to do this:

Locate the method in the Solution Explorere and right-click on it and select 'Debug Method':

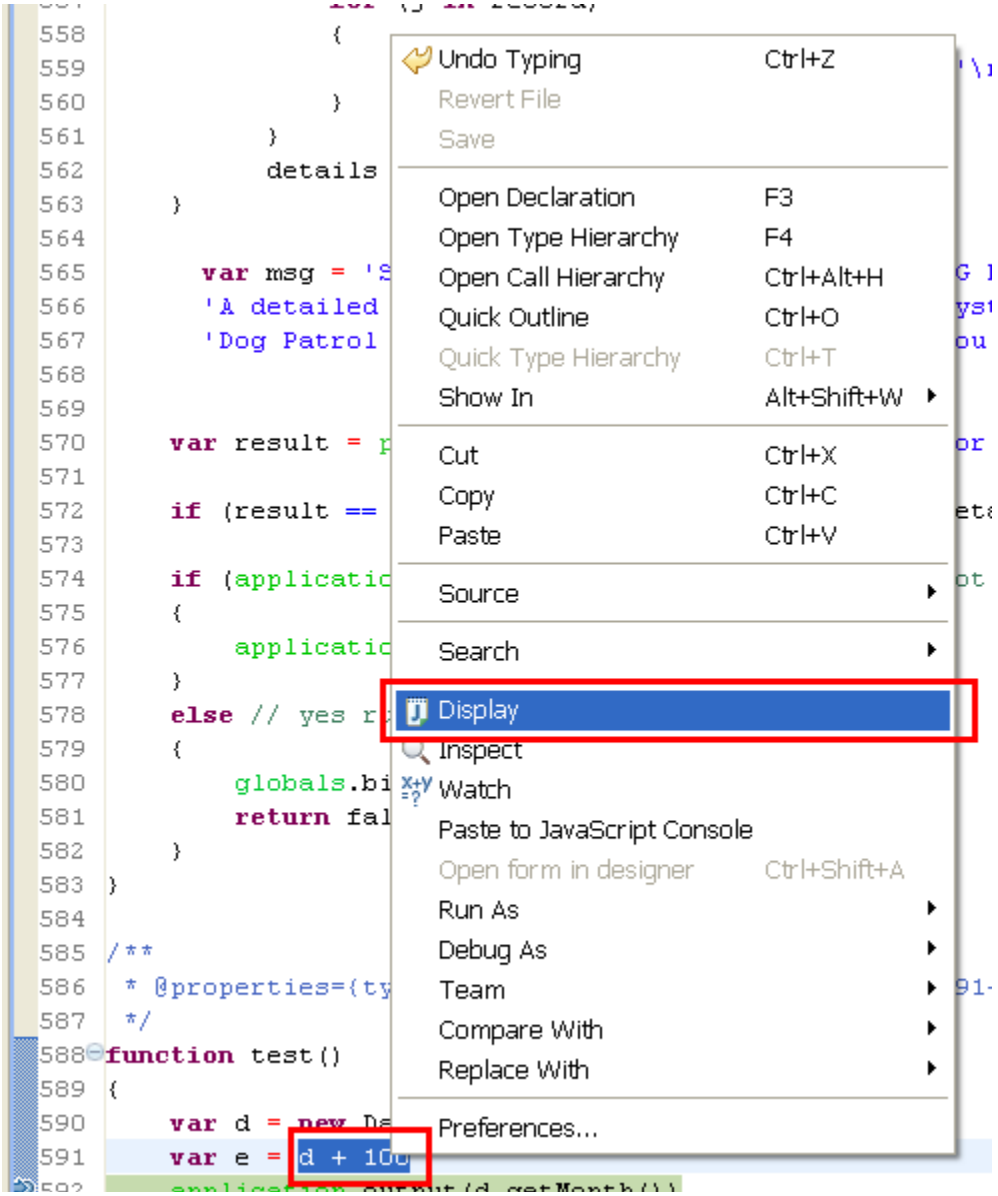


Or open the method in the script editor and locate the method in the Outline tab which lists all the functions in the Javascript file currently being edited. Right-click on it and select 'Debug Method':



Inspecting variables, expressions, objects in your code

When you have placed a breakpoint in a method and execution has halted, you can select any variable, object or expression in you code to see what it currently evaluates to. For a single-word item such as a variable or dataprovider, just hover over the word in the editor and the value should come up in a tooltip. For longer expressions you need to select the expression, right-click and select 'Inspect' or 'Display'.



Debugger behavior with showFormInDialog()

If you are using the debugger to step through a method that has a modal showFormInDialog command in it, you'll get some strange stepping behavior when you get to the showFormInDialog() command itself. If you're debugging a method with a showFormInDialog() call, I'd avoid using the debugger's step button and use application.output() instead.

by Adrian McGilly, Servoy Consultant & Trainer