Defining a Data Model

In This Chapter

Data Providers

Servoy's 'Data Providers' are the atomic units of the data binding layer. A Data Provider holds an individual data value which may be bound to both User Interface elements, as well as back-end resources. Thus, Data Providers are the link between the user experience and the raw data.

Types of Data Providers

There are several types of Data Providers, each with different data bindings and uses:

- Columns
- Calculations
- Aggregations
- Global Variables
- Form Variables

Once established, a data provider can be bound to user interface elements at design-time. At run-time, the elements come alive with data which is contextual to the application state and the Data Provider's back-end bindings. The most obvious example is a text field which is bound to a column in the database table. However, any of the types of Data Providers can be bound to any UI components, including fields, labels, buttons, tooltips, etc.

Columns

Database Columns are the most common type of Data Provider in Servoy. When a named server connection is established, the entire database structure - all tables and their columns - is read. Columns become available vehicles for data binding. Additional metadata properties are used to further describe how each column is treated in the application.

Calculations

A Calculation is very much like a database column, except that its value, rather than being stored, is dynamically computed each time it is requested.

See Calculations for more information on how to work with calculations.

Aggregations

An aggregation is a data provider which represents a database column that is aggregated over a set of records.

See Aggregations for more information on the properties of an aggregation and how to use it in Servoy.

Variables

Variables are Data Providers which, unlike columns, calculations aggregations do not bind to any persistent, back-end data source. Instead, variables store data in-memory for the duration of the client session only. However, variables may bind to UI components the same as any other data provider.

Data Types

While variables can actually contain any JavaScript object or literal value, they must initially belong to one of the following data types:

- Text
- Integer
- Numeric
- Data/Time
- Media

⚠ Notice that these are the same as the generalized data types for column-based Data Providers.

Scope

Variables can belong to one of two scopes:

- Global
- Form

Other than programming scope, the only difference between the two is that globals can be used as keys in relations.

Relations

Relations, at design-time model associations between two tables by joining on key data providers. At runtime, a relation becomes a context-sensitive programming reference to related data. Thus, relations are simple, but powerful mechanisms to display, edit and search for data from any context. They can be used, not only to model simple database relations, but also to create sophisticated filters and searches.

Design-Time Properties of Relations

Relations have several design-time properties that dictate how related foundsets will behave at runtime.

Source Table

Can be any database table or view from any named server connection.

At runtime, a related foundset will exist in the context of a single record from the source table. For example, the relation *customer_to_orders*, will become available in the context of any record in a foundset which is based on the *customers* table.

Destination Table

Can be any database table or view from any named server connection and is not limited to the same database as the destination table.

At runtime, a related foundset will contain records from the destination table

⚠ The destination table can exist in a separate database from the source table. This is a powerful feature, but it is worth noting that a related foundset, who's relation is defined across two databases will not be available when the source foundset is in find mode. This is because a related find requires a SQL JOIN, which cannot be issued across databases for all vendors.

Relation Items

The nature of the relation between the source and destination tables is defined by one or more Relation Items. Relation Items are expressions, each consisting of a pair of key data providers (one from each table) and a single operator.

Source Data Provider	Operator	Destination Data Provider

The Relation Items will be used to constrain the records that are loaded in the related foundset, such that records are loaded only when **all** of the expressions evaluate to be **true**.

Example: This example creates a relation between the *customers* and the *orders* table. A related foundset will only load *orders* records with a *customerid* equal to the *customerid* in the context of the source *customer* record.

Source (customers table)	Operator	Destination (orders table)
customerid	=	customerid

Data Providers

One data provider from each table will serve as an operand in the key-pair expression. Therefore, both data providers must share the same data type. Columns, calculations and global variables may all be used as the source data provider. However, only columns may be used for the destination data provider.

Source Data Provider - Available Types

- Columns
- Calculations
- Global Variables

Destination Data Provider - Available Types

- Columns Only

Related foundsets are loaded in the context of a single source table record, which is already known. Therefore, any global variables, as well as the source record's calculations can be evaluated and used as a key. However, only columns from the destination table can be used as the dynamic data providers cannot be evaluated on behalf of destination records before they are loaded.

Operators

Each key pair expression is evaluated using a single operator. Certain operators are only applicable to certain data types. Below is a list of all available operators and the data types for which they are applicable.

Operator	Description	Data Types
=	Equals	Text, Integer, Number, Datetime
>	Greater Than	Text, Integer, Number, Datetime
<	Less Than	Text, Integer, Number, Datetime
>=	Greater Than or Equal To	Text, Integer, Number, Datetime
<=	Less Than or Equal To	Text, Integer, Number, Datetime
!=	NOT Equal To	Text, Integer, Number, Datetime
like	SQL Like use with '%' wildcards	Text
not like	SQL Not Like use with '%' wildcards	Text
#=	Case-Insensitive Equals	Text
#!=	Case-Insensitive NOT Equal To	Text
#like	Case-Insensitive SQL Like	Text
#not like	Case-Insensitive SQL NOT Like	Text
^||=	Null OR Equals	Text, Integer, Number, Datetime
^||>	Null OR Greater Than	Text, Integer, Number, Datetime
^||<	Null OR Less Than	Text, Integer, Number, Datetime
^||>=	Null OR Greater Than or Equal To	Text, Integer, Number, Datetime
^||<=	Null OR Less Than or Equal To	Text, Integer, Number, Datetime
^||!=	Null OR NOT Equal	Text, Integer, Number, Datetime
^||like	Null OR SQL Like	Text
^||not like	Null OR SQL NOT Like	Text
^||#=	Null OR Case-Insensitive Equals	Text
^||#!=	Null OR Case-Insensitive NOT Equals	Text
^||#like	Null OR Case-Insensitive SQL Like	Text
^||#not like	Null OR Case-Insensitive SQL NOT Like	Text

Text-Based Expressions

Expressions which contain the *SQL Like* or *SQL NOT Like* operators should be used in conjunction with values that contain wildcards (%).

```
customers.city like New%         // Starts with: i.e. New York, New Orleans
customers.city like %Villa%         // Contains: i.e. Villa Nova, La Villa Linda
customers.city like %s              // Ends with: i.e. Athens, Los Angeles
```

Join Type

A relation can specify one of two SQL Join Types. A SQL join used when a **find** or a **sort** is performed using related criteria and thus, the join type will affect behavior in these situations.

Inner Join

SQL Inner Join does not return any rows for parent records which have no related records. Therefore, if a sort or a find is performed when a related data provider is used for criterion, the related foundset may have records omitted due parents with no child records.

Left Outer Join

SQL Left Outer Join will return always return a row for the parent record even if there are no related records. Therefore, if a sort or a find is performed when a related data provider is used for a criterion, the related foundset will include all matching records, regardless of the presence of related records.

Example: Assume that the user chooses to sort a customer list containing 50 records. The sort is based on the account manager's last name, which is in the *employees* table. However, 3 of the customers don't have an employee listed to manage the account.

```
foundset.sort('customers_to_employees.last_name asc');
foundset.getSize(); //        returns 50 if the customers_to_employees relation specifies left outer join, 47 if
the relation specifies inner join.
```

Initial Sort

Foundsets, including related foundsets, have a sort property. By default, any foundset is sorted by the primary key(s) of the table upon which it is based. Relations have an *Initial Sort* property, which overrides the default sort, such that any related foundset is initialized to use the sorting definition defined by the relation object. For more information see foundset sorting.

Deprecated

A relation can be deprecated, and a description has to be provided to hint users about what the alternative is.

Encapsulation

A relation has encapsulation property, similar to the form encapsulation property.

- Public – accessible from everywhere
- Hide in Scripting; Module Scope – code completion is disabled for the relation, and it is accessible only from the module that it was created in
- Module Scope – accessible from the module it was created in

For non-public encapsulation, if the relation is accessed from somewhere else, you get a build marker in Problems View, but it will still function properly.

Referential Integrity Constraints

Relations have three options that support referential integrity in the data model. These options control both the actions that are permissible, as well as cascading actions in the data model.

Allow creation of related records

This option is enabled by default and it specifies that records can be created within a related foundset. Moreover, when records are created in a related foundset, the key columns in the new record may be automatically filled with the corresponding values from the source record.

Example: Assume a relation, *customers_to_orders* defined by a single key expression, *customers.customerid = orders.customerid*

```
customerid;                       // 123, the customer's id
customers_to_orders.newRecord();// create the new record
customers_to_orders.customerid;        // 123, the order record's foreign key is auto-filled
```

Key columns will be auto-filled for expressions using the following operators:* =

- #=
- ^||=
  If this option is disabled, then records cannot be created in a related foundset. If attempted a ServoyException is raised with the error code, NO_RELATED_CREATE_ACCESS.

Allow parent delete when having related record

This option is enabled by default. When disabled, it will prevent the deleting of a record from the source table if the related foundset contains one or more records. If the delete fails, a ServoyException is raised with the error code, NO_PARENT_DELETE_WITH_RELATED_RECORDS.

Example: Assume the relation *customers_to_orders* has disabled this option. An attempt to delete a customer record will fail, if that customer has one or more orders.

Delete related records

This option specifies that records in a related foundset can be deleted. Moreover, it also enforces a cascading delete, such that when a source record is deleted, all records in the related foundset will also be deleted, eliminating the possibility of orphaned records.

Example: Assume the relation *customers_to_orders* has enabled this option. The deleting of the customer record will cause all related order records to be deleted.

Special Relations

Dynamic Relation

In addition to database columns, calculations and global variables may be used as keys for the source table. This provides a means to implement dynamic data filters without writing any code or SQL. A related foundset is refreshed whenever the value of a source key changes. Thus, by using variables and calculations as keys, developers can articulate nuanced views of data that are contextual not only to the source record, but also the changing state of the application.

Example: Assume that one wants to filter a customer's orders by date in different ways, i.e. today, this month, last month, this year, last year, etc. One could define the following relation from *customers* to *orders*.

Source	Operator	Destination
customerid	=	customerid
globals.orderFilterStart	<=	orderdate
globals.orderFilterEnd	>=	orderdate

By simply changing the value of the global variables (either programmatically or through the GUI), the related foundset for a customer's orders is updated instantly.

Global Relation

Global relations are simply relations that use only global variables for source data providers. The key difference between global relations and regular table relations is that the related foundset exists in a global context, having no source record as a context. The obvious benefit is that the globally related foundset will be available ubiquitously, instead of being limited to the context of records based on a source table.

Example: Assume that a customer service rep should have a dashboard of all of their orders that are due today. This view could be accomplished using a global relation on the *orders* table, which could be used anywhere in the application, such as a form in tabpanel to show a dashboard view.

Source	Operator	Destination
globals.currentUserID	=	sales_rep_id
globals.today	=	orderdate

Self Relation

Relations may have the same source and destination table. This is called a Self Relation and has a variety of applications, such as showing data which is hierarchical in nature, or simply showing other records in the same table, which have similar attributes.

Example: When looking at a particular order record, the user may like to see a portal containing a list of all of the other orders made by the same customer as the current order. This could be expressed using a Self Relation, *orders_to_orders_by_customer,* containing two relation items. The first specifies the same customer, the next ensures that the current order is omitted from the foundset.

Source	Operator	Destination
customerid	=	customerid
orderid	!=	orderid

Example: An *employees* table is organized such to reflect a companies chain of command. When looking at an employee record, a user should be able to easily see the employee's boss, as well as the people that the employee is managing.

employees_to_employees_manager: The current employee's boss

Source	Operator	Destination
manager_id	=	employee_id

employees_to_employees_managing: The employees managed by the current employee.

Source	Operator	Destination
employee_id	=	manager_id

Container Relation

Container relations are a type of Self Relation used to reference the source record's foundset. To create a container relation, the following must be true:

- Source table and destination table are the **same**
- There are **no** relations items

A container relation is applicable when a parent form must contain a child form in a TabPanel and the child form is based on the same table and should show the same foundset as the parent. Use of a container relation will ensure that the parent's foundset is shared with the child.

Example: There are three forms based on the customers table: *customerMain*, *customerList*, *customerDetail*. The *customerMain* form contains the other two forms in a single, unrelated TabPanel, such that the user can easily toggle between list and detail view. The approach will work so long as the *customerMain* form is also using an unrelated foundset. However, if the *customerMain* form loads a related foundset (i.e. it is shown through a relation), then the two child forms will still be unrelated and therefore out of sync. The solution is to create a container relation for the customers table and show the 2 child forms through this relation. This guarantees that they will always share the same foundset.

If the child forms are not shown through a relation, the approach may still work in many cases, because unrelated forms of the same table will share a single foundset. This is discussed in more detail in the section covering foundsets.

Value Lists

A Value List is a powerful data modeling feature which, at design-time models a particular list of values, which may be static, or dynamic, data-driven lists. Value lists may be bound to UI components, as well as interacted with programmatically. At run-time, a value list returns a list of dynamically generated values for the context of a form/foundset. When bound to a UI component, a value list is displayed as a list of choices, i.e. in a combobox or radio button group, etc. Value lists obviate the need to write code and SQL, thus greatly enhancing developer productivity.

Display Value

The value that is shown to the user, but may not necessarily be stored in a data provider.

Stored Value The value that is returned into a data provider to which the value list is bound.

Types of Value Lists

When creating a Value List, a developer will specify one of four types of value lists, each having different properties and applications.

Custom Values

This is the simplest type of value list. It represents a static list of available values, both displayed and stored. A developer hand-enters values directly into the value list editor. The displayed and stored values may be literal, or evaluated at runtime using i18n keys and global variables.

Example In its most basic form, the value list is simply a list of values from which to choose. The values which are displayed are also stored into a data provider.

Yes
No

Example When designing a value list, a developer may use a pipe character '|' to separate **display** values from **stored** values. In this example, the options *Yes* and *No* will be displayed to the user, but the values *1* and *0* will be respectively stored into a data provider.

Yes|1
No|0

Example Value lists can easily be made multi-lingual by using i18n keys in lieu of literal values. Here the same value lists is made multi-lingual and the *Yes* and *No* values will be displayed in the language of the user's locale.

i18n:yes|1

i18n:no|0

Example Value Lists can also evaluate *Data Tags* to store literal values that are already declared as global variables. This approach is recommended to store constant values, which are declared once in the entire application. Here a list of order statuses are used. The display values are made multi-lingual as above. And the stored values reference global variables defining constant values for each order status. The data tag takes the form *%%globals.myVariableName%%*.

i18n:orderStatusNew|%%globals.ORDER_STATUS_NEW%%
i18n:orderStatusConfirmed|%%globals.ORDER_STATUS_CONFIRMED%%
i18n:orderStatusShipped|%%globals.ORDER_STATUS_SHIPPED%%

Global Method Values

A value list can be bound to a global method which supplies the values every time the value lists is used. This option gives the developer the most control, but also requires that the developer write code, and is therefore recommended to be used when the other value list types are not sufficient. The method is invoked often, each time the value list is referenced and each time the context changes.

Parameters

String displayValue - The value entered by the user into the field bound to the value list. This value will be null, unless the user is typing into the field, in which case the method may be called often as the user types. This parameter is useful when used in combination with Type Ahead fields, allowing the developer to filter the values as the user types.

Object realValue - The real value that is stored in a data provider that is bound to the field for which the value list is invoked. This value is passed in to allow developers to provide display values for a given real value which may not already be in the list.

JSRecord record - The record which is the context for the value list.

String valueListName - The name of the value list for which values will be supplied as multiple value lists could use the same method to obtain their values.

Returns

JSDataSet containing the values, both displayed and stored. The dataset should have two columns, display and stored respectively. If the dataset has only one column, then it will be used for both displaying and storing values.

Table Values

A value list can be derived from all of the values in a single table. This approach is ideal to use a real table from which to look up values. The following properties apply to Table-based value lists:

Table The database table from which the values are drawn.

Stored Value This specifies the name of the column that will provide the values at run-time.

Display Value This specifies the name of the column(s) that may provide value which will be displayed when the value list is bound to a UI element. A value list may specify up to three display columns.

Separator Character This specifies a String which will separate display values when multiple display columns are specified. i.e. ', ' could be used to separate the database columns *last_name*, *first_name*

Value List Name as Filter This setting specifies that a column, named *valuelist_name*, will be used to filter the records used in the valuelist . If specified, only those records, who's value for the *valuelist_name* column equals the name of the value list itself will be returned.

> ⚠ It is not common to use the Value List Name as Filter setting, unless a reusable, generic table is used to hold many display/values for different value lists.

Related Values

Related value lists are similar to table-based value lists The only difference is that the table which is used is filtered by the characteristics of a relation. Moreover, the relation itself will be contextual to the form/foundset for which the value list is invoked.

Example A value list *project_people* is based on the relation *projects_to_people*. Therefore the values will be derived from the right-hand *people* table. And the value list may be used in context of the left-hand *projects* table. The resulting list of people will be contextual to the selected project record of a form /foundset.

Relation-based value lists can traverse across as many relations as need be to arrive at the destination table.

Example In the above example, a simple one-to-many relation *projects_to_people* was used. Suppose however that there is a many-to-many relation between projects and people expressed by a link table *project_people*. The value list could traverse across two relations to return the correct values: *projects_to_project_people.project_people_to_people*. The far right-hand table, people, is still used to return values in context of a single project record.

Design-Time Properties of Value Lists

There are several design-time settings available for every value list

Fallback Value List

This property specifies another value list which may be used in the event that a record's value does not fall within the set of values provided by the value list. Fallback value lists are useful in find mode.

Example Using the above example for the project_people value list, which shows a list of people related to a project; Imagine that a person was removed from a project, however, the value stored in a related records may still point to that person. In this case, the value would no longer show up in a bound component (i.e. combo box, radio buttons, etc.) as the person is no longer a valid selection. Nevertheless, the person is still referenced by the record. Therefore it may be advantageous to use a *fallback* value list, say one that displays all people in the people table, to ensure that the person is displayed. However, when the record is edited, only the values in the *project_people* value list will be displayed.

Allow Empty Value

This is a simple setting to indicate if a value list will have an empty/ null value available for selection.

Sorting Definition

The values contained in a table-based, or relation-based value list can be sorted on any columns in the table. Additionally, any related columns may be used as well.

Deprecated

A value list can be deprecated, and a description has to be provided to hint users about what the alternative is.

Encapsulation

A value list has encapsulation property, similar to the form encapsulation property.

- Public – accessible from everywhere
- Module Scope – accessible from the module it was created in

For non-public encapsulation, if the value list is accessed from somewhere else, you get a build marker in Problems View, but it will still function properly.