Personal Information Manager Overview & Installation Guide

Intended Audience

This article and starter kit are aimed at medium and advanced level Backbase developers. It is assumed that you already have hands-on experience with XHTML, CSS, BXML and XPath, and that you have played around with the Backbase controls. Also, to run the application you need access to a server with PHP and MySQL. Installation guidelines can be found at the end of this article.

The following topics will be covered in this article and in the starter kit code: combining controls, connecting the GUI to the server and in-depth data binding.

All my contacts in one place: online

This is an application for managing personal information in general. It has an implementation of the vCard standard for managing contact information. It showcases a common way of managing information by exposing it on different levels and in alternative views. The contact information is stored in a database on the server and you can add, edit and remove both groups and contacts.

As you will see, this application does not look like a web page at all, even though technically it is. In the Personal Information Manager, we have focused on creating the look and feel of a traditional desktop application that any user will be instantly familiar with. But, because it is a RIA (Rich Internet Application), it is accessible on the web and does not need to be installed by the user: starting to use it is as easy as surfing to the web address where it has been deployed.

On the surface: the GUI

The screen space is partitioned with a series of panels and panelsets that define the basic layout. In the top panel you will find a toolbar, and on the left side there is a navpanel with a tree inside of it. The tree is used to select groups of contacts, effectively offering a high-level categorization of the information in the database.

The look of the remaining space is determined by which view has been selected: the card view or the list view. In the card view, the contacts in the currently selected group are displayed as cards. The cards have been arranged in columns by putting them in a tilelist control. In the list view, the

contacts are displayed as a list of row items, with labels for each column: Name, Organization, Phone number and Email address. The listgrid control is used to offer the contacts overview in this format.

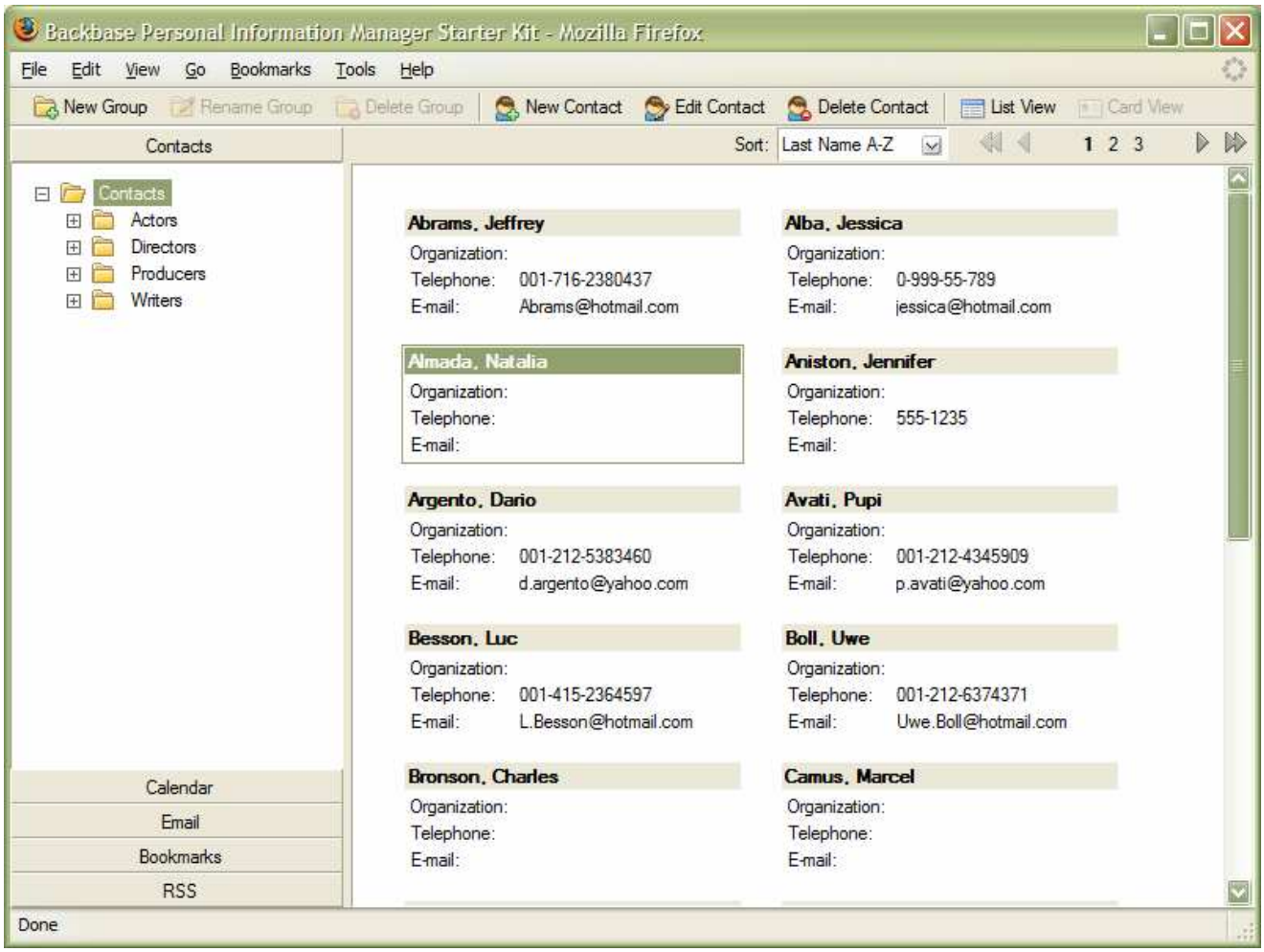
Both the tilelist and the listgrid manage the potentially large dataset with the use of pagination: they show only a part of the dataset at a time. This is a convenient way of avoiding the performance slowdowns that would occur when loading and displaying thousands of items at once. In order to navigate the pages in the tilelist and listgrid, a pager control is used. The pager control has buttons for going to the first, the previous, the next and the last page. It also displays some numbered buttons that indicate at which page you are.

In the card view, there is a select box next to the pager control, which controls the field on which the data is sorted and whether the sort order should be ascending or descending. In the list view, sorting is controlled by clicking on the column headers.

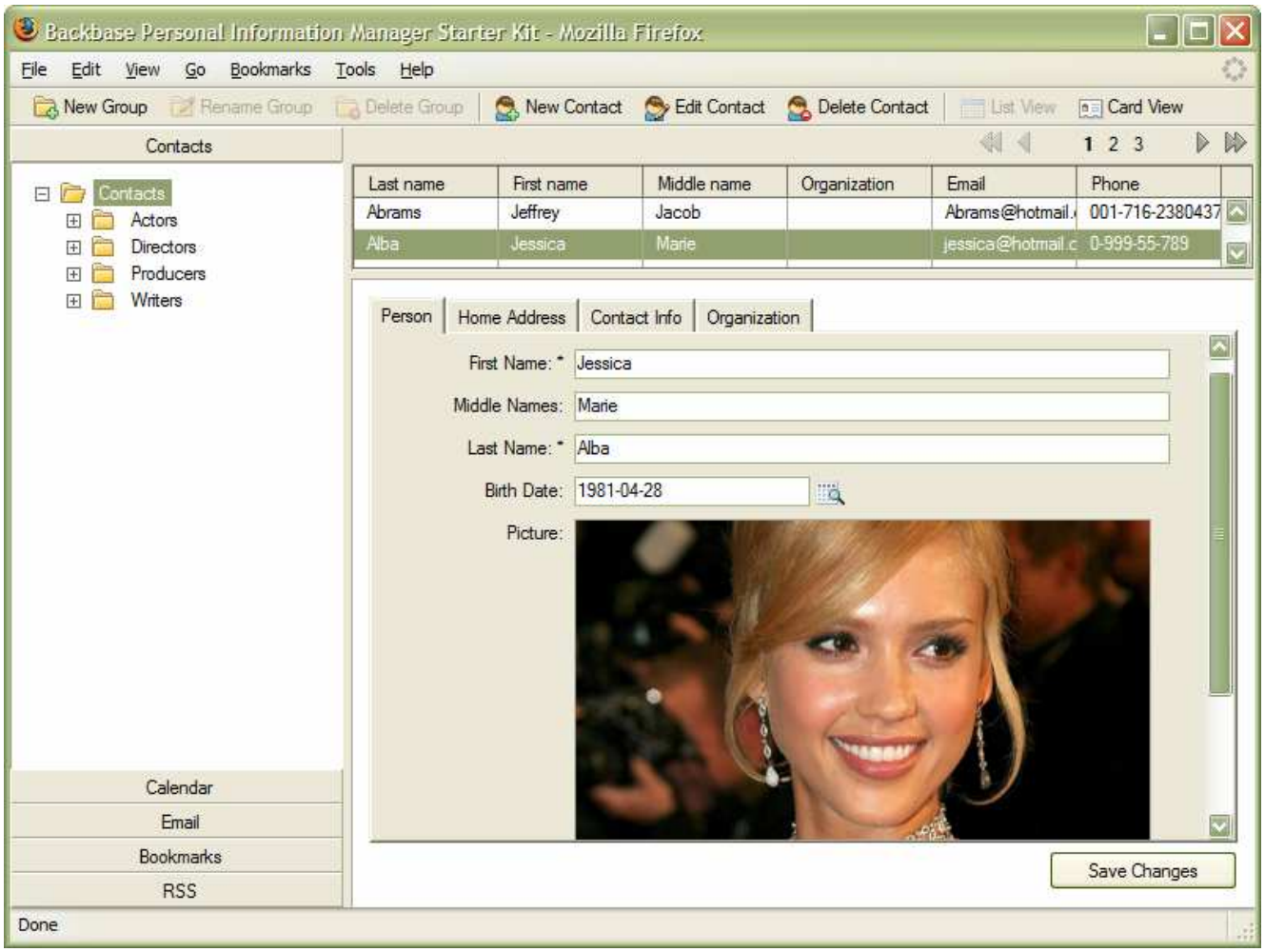
The detailed information of each contact becomes available when selecting a contact or double-clicking it. This gives you access to over 25 fields like date of birth, home address and different email addresses.

In the list view a tabbox control will be enabled and populated with information when you select a row in the listgrid. In the card view a modal dialog with a tabbox inside of it will be opened when you double click on a card.

In the following screenshots you will recognize the GUI controls that have been mentioned.



The card view



The list view

Under the hood: the startup file

If you look at the code in the index.html file, you will recognize the panelset and the main GUI structures in it. You will not find all elements of the GUI in there, because some of them are only loaded when necessary. For example, on startup the tilelist will be loaded separately only if the card view is selected. If the list view is selected, the listgrid and tabbox are loaded. The modal dialogs are also only loaded when necessary, and the tree for selecting contact groups uses its internal lazy loading implementation for loading its children.

Apart from the GUI structure, you will find some small bits of logic in the startup file:

- In a small xmp element before the main xmp element, an s:execute statement configures some user settings with cookies. It also sets the value of some global variables that are provided by the BPC. The code in the first xmp element will be executed completely before the main xmp element is even parsed, which makes it suitable for

adjusting settings that will be used in the main application.

- In the main xmp element, an s:include element loads the definition of the custom b:card control. This is not necessary for the definitions of the standard controls, because those get loaded automatically when necessary.

- There is also an s:include for loading the main application logic, which consists of a set of controllers and behaviors. This logic does not need to be in a separate file, but it makes it easier to edit them and also keeps the separation of presentation from logic structures clear.

- An s:execute statement kick starts the application logic, which handles for example which view is visible. This will run after the GUI controls have been constructed in the BXML tree and the XHTML DOM tree, so after every element has finished its construct event.

Architectural patterns of the main application logic

The main client side application logic is in the application_logic.xml file. It is split in three controllers that centralize the application event handlers and some small behaviors that capture events in the GUI controls.

The controllers are constructed by defining a div element with display:none in its style attribute, because it should never displayed. Nested in the controller are the event handlers. There are several reasons for choosing this setup.

Controller event handlers

By nesting the event handlers in separate controllers, we are effectively creating an object oriented control structure. This helps avoid using cumbersome global names for events such as "group-edit-confirm-modalmode-listview". Instead, names like "confirm-edit" can be used both in the group controller and the contact controller.

You can trigger an event in a specific controller with the trigger command:

```
<s:task b:action="trigger" b:event="request-save" b:target="id('contact-controller')" />
```

Controller variables

Because the controller is an element, it can have tag scope variables to remember things like view state and selections. Using tag scope variables avoids the conflicts inherent in using global variables, and effectively isolates parts of the application from each other.

You can access tag scope variables in a specific controller element by first writing the path to that element, and then the variable name:

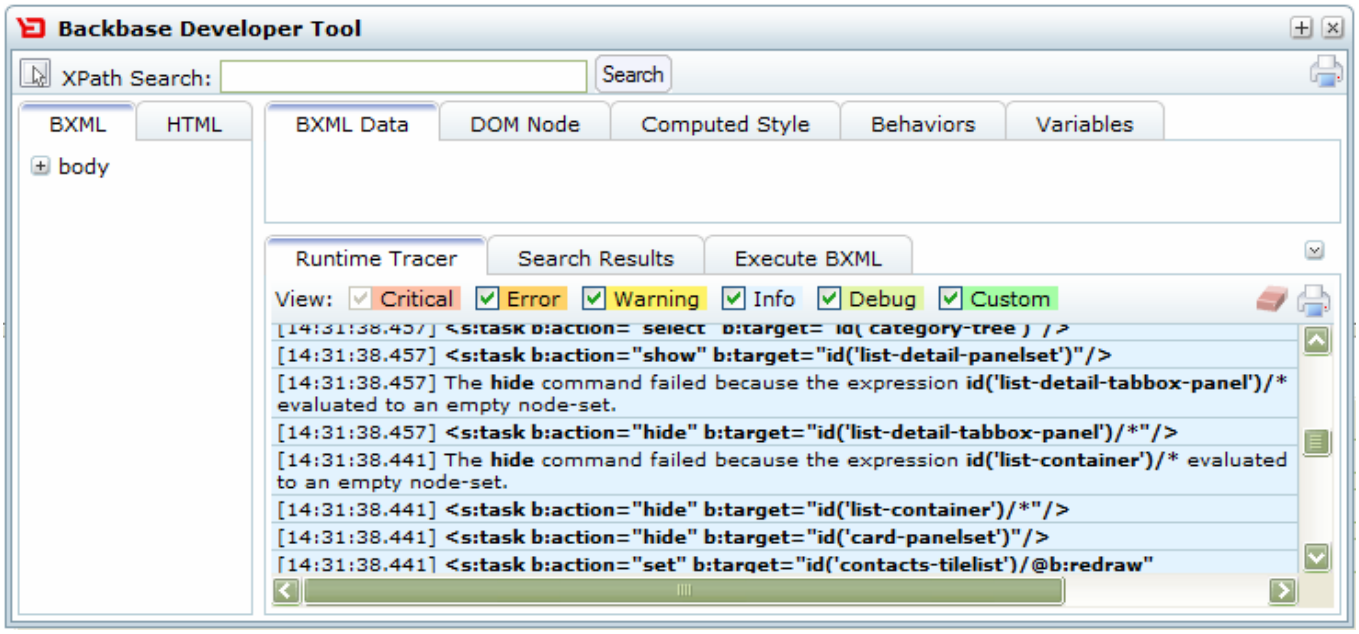
```
<s:task b:action="remove" b:target="id('contact-controller')/$selected-contact" />
```

Analyzing event execution

Events can be triggered from the GUI as well as from other event handlers. It can sometimes be tricky to find out when an event is triggered and in what order the event handling takes place. There are some different approaches to getting information about event and task execution, and I will describe two methods here. Both methods involve using the Runtime Tracer in the Backbase Developer Tool, which you can open if you are using the developer build of your Backbase installation. Open it by pressing Alt+1, or select it from the Tools menu which becomes visible when you press Escape.

Debugging with the debug attribute

Set the attribute b:debug=”true” on an element, like the view controller. Now when you look at the Runtime Tracer in the Backbase Developer Tool, you will notice that you get a message for every task that is executed in the context of the view controller.



Debug messages in the Runtime Tracer

This level of detail can be nice, but seeing every executed task is usually overkill if you just want to track the order in which event handlers get triggered.

Debugging with custom trace messages

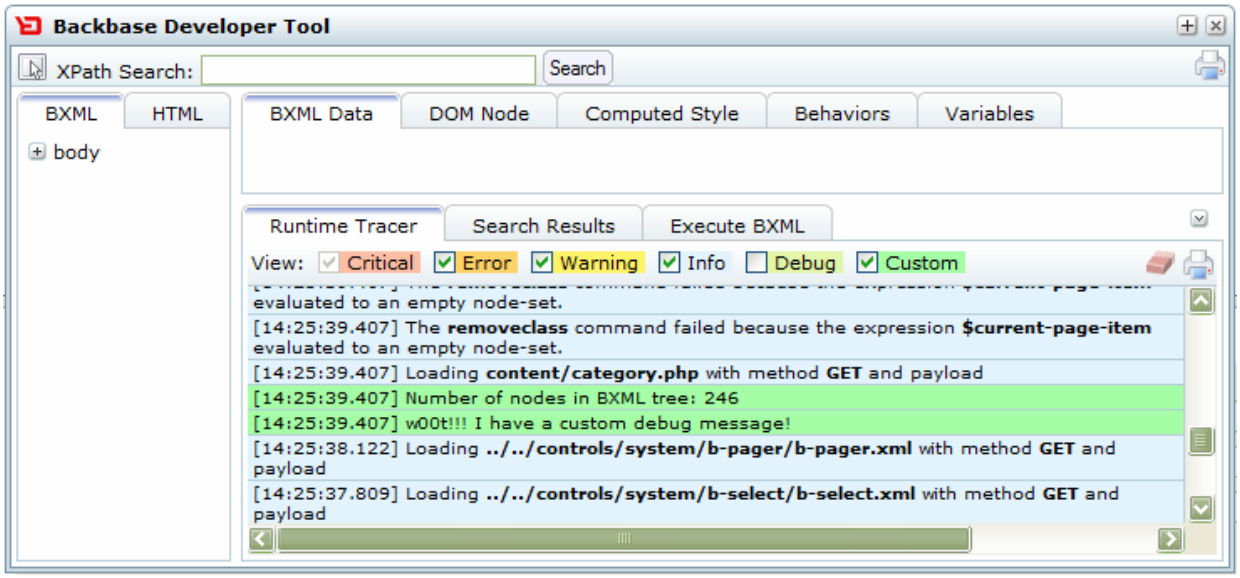
By using the msg command, you can create your own debug messages in the Runtime Tracer. By putting debug messages in event handlers, you can see whether they get triggered or not, and in what order. A line of code that creates a message at custom level would look like this:

```
<s:task b:action="msg" b:value="w00t!!! I have a custom debug message!" b:level="custom" />
```

If you want to use an XPath expression in the b:value attribute, add curly braces around it:

```
<s:task b:action="msg" b:value="{concat('Number of nodes in BXML tree: ', count(//*))}" b:level="custom" />
```

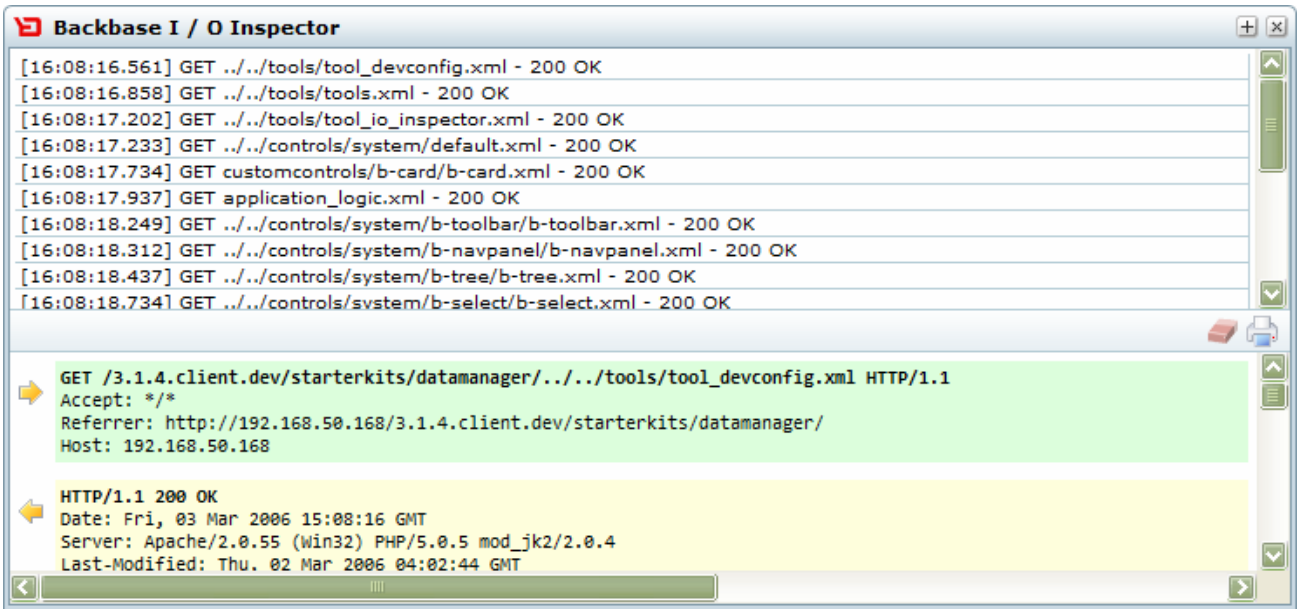
The result of these msg commands will look a lot like this:



Custom messages in the Runtime Tracer

Population of Groups and Contacts: browser perspective

The groups and contacts information is dynamically composed on the server. Whenever a piece of information is required in the browser, it is requested, received and displayed. To analyze the request and receive process, use the Backbase I / O inspector. You can open it by pressing Alt+2, or selecting it from the Tools menu that is shown when you press Escape.



The I / O Inspector shows all load and send transactions with the server

Population of Groups: the tree control

When a folder in the b:tree control is opened, its internal lazy loading mechanism is triggered. It requests the data for building its children, and receives an s:render block for every child. The s:render blocks are contained in an s:execute block, which is the root element in a valid XML document that will be accepted by the Backbase Presentation Client (BPC) and executed.

A request / response for populating the tree control

Population of Contacts: the listgrid control

The b:listgrid control can be populated with data in XHTML format as well as data in raw XML format. There is not a large performance difference, because the native browser XSLT engine is used for transforming the raw XML to XHTML, and that engine is fast.

For the purpose of demonstration, we chose for using raw XML in this starter kit. This means that it is necessary to also provide XSL stylesheets for doing the transformation: one for modern browsers, and one for Internet Explorer 5.0 / 5.5. Those older versions of Internet Explorer were released with a non-standard XSLT engine.

To get all the data binding details of this application working correctly, we slightly changed the out-of-the-box XSL stylesheets that come with the controls. For example, we added an extra attribute called contactid to be able to identify the items in the listgrid.

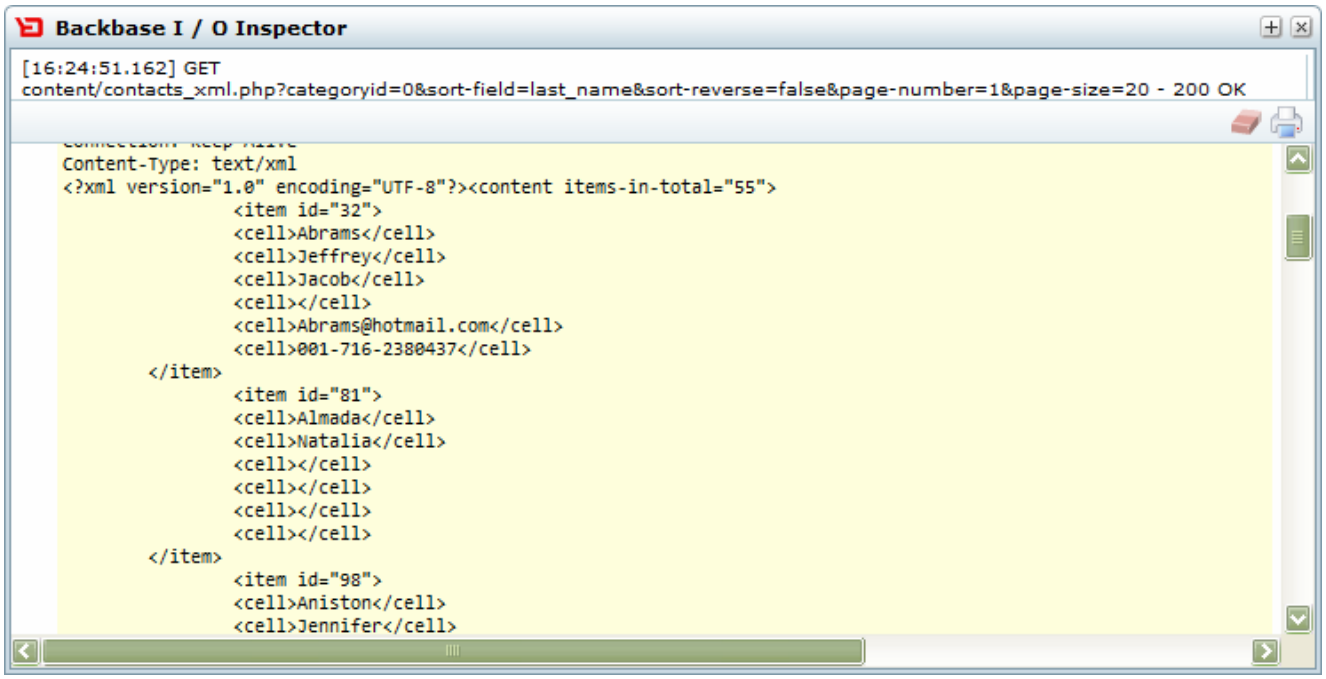
The basic structure of the raw XML is a root element with the attribute items-in-total, and a child element for every contact. The items-in-total attribute is a number representing the total number of contacts in the database, and is needed by the b:pager control for calculating the total number of pages. Each of the elements representing a contact have an id attribute and a child element for every column in the listgrid. The name of the XML elements is irrelevant, what matters is their structure, their content, and their attributes.

When the XML is requested, a set of name / value pairs is passed along to tell the server exactly what data is needed. The b:url attribute of a listgrid may already contain some of these name /

value pairs, and the listgrid automatically adds page-number and page-size information to the url when requesting a new page.

In the Personal Information Manager starter kit, the following parameters are used when requesting a page for the listgrid:

- **categoryid** specifies which group of contacts is selected
- **sort-field** specifies on which field the data should be sorted
- **sort-reverse** specifies whether the data should be in ascending order or reversed
- **page-number** specifies which page is needed (added automatically by the listgrid)
- **page-size** specifies the requested number of items per page (added automatically by the listgrid)



A request / response for a listgrid page

Population of Contacts: the tilelist control

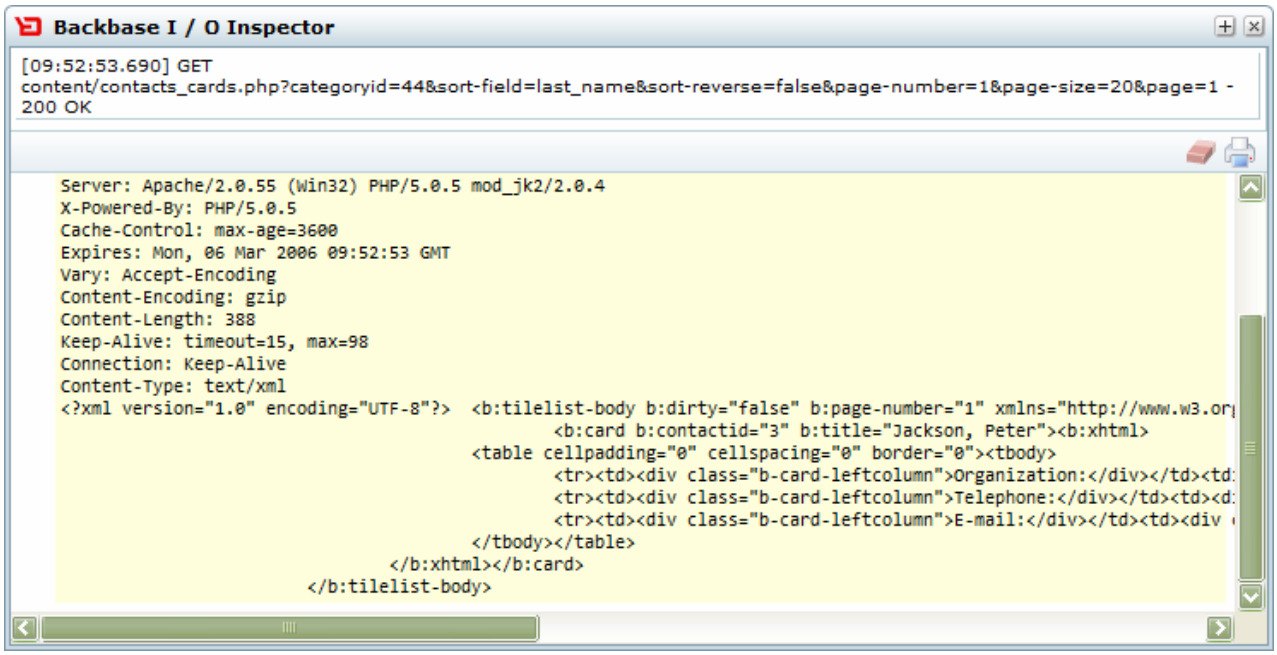
The tilelist control is different from the listgrid control in the type of data that it accepts. It only accepts BXML, and the root element must be a b:tilelist-body element. The BXML does not need to be transformed with XSLT. The children of the b:tilelist-body element will become the tiles.

In this starter kit, we have chosen for a custom control called the b:card that is used as the tiled element. The custom b:card control has been optimized specifically for this application; it uses some application-specific events. Also, part of its content actually does not consist of custom BXML elements, but of XHTML markup contained in a b:xhtml tag. Encapsulating XHTML markup in a b:xhtml tag ensures that the contents will not become nodes in the BXML Tree, they are only constructed in the browser's DOM Tree. This is a useful design pattern for repetitive data

structures that don't really need to be manipulated as BXML nodes. It saves potentially dozens or even hundreds or thousands of nodes from being constructed in the BXML Tree. That's a very useful optimization because to maintain snappy performance you don't want the BXML Tree to grow much bigger than 500 nodes, even though its size is unlimited.

In the Personal Information Manager starter kit, the following parameters are used when requesting a page for the tilelist:

- **categoryid** specifies which group of contacts is selected
- **sort-field** specifies on which field the data should be sorted
- **sort-reverse** specifies whether the data should be in ascending order or reversed
- **page-number** specifies which page is needed (added automatically by the tilelist)
- **page-size** specifies the requested number of items per page (added automatically by the tilelist)



A request / response for a tilelist page

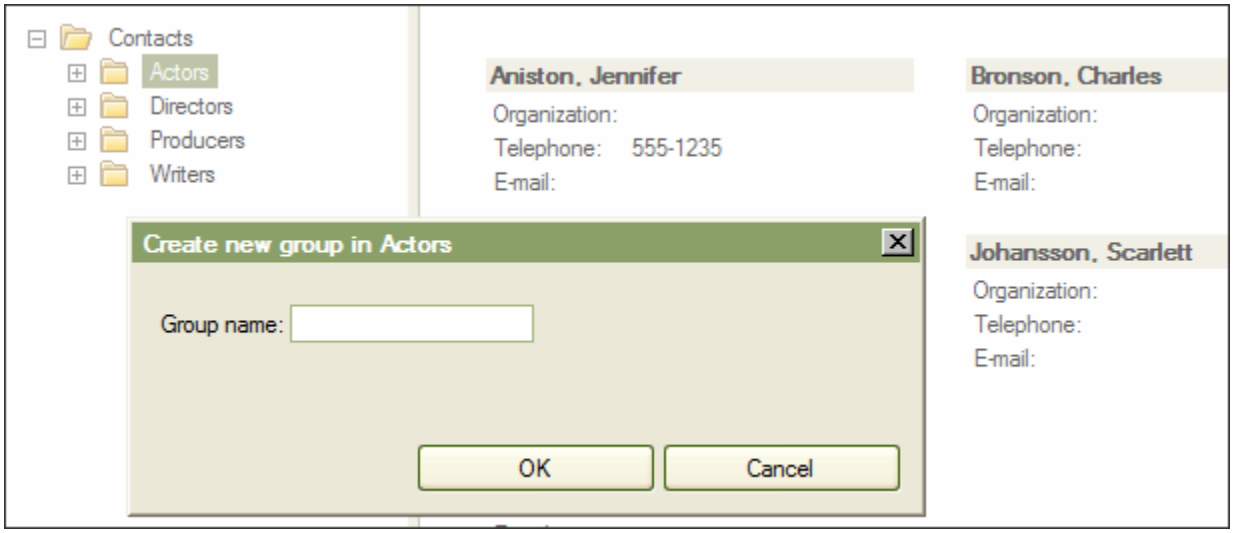
Creating, renaming and deleting Groups

Creating, renaming and deleting contact groups is done with toolbar buttons that trigger a modal dialog. The same modal dialog is reused every time, so it only needs to be loaded and constructed the first time it is used.

Every time a change is made, the server is notified of the change, and the change is saved in the database. When a new group is created, the tree element representing its parent will be opened.

This triggers its internal lazy loading mechanism, requesting all children from the server. The client display gets updated by constructing new nodes.

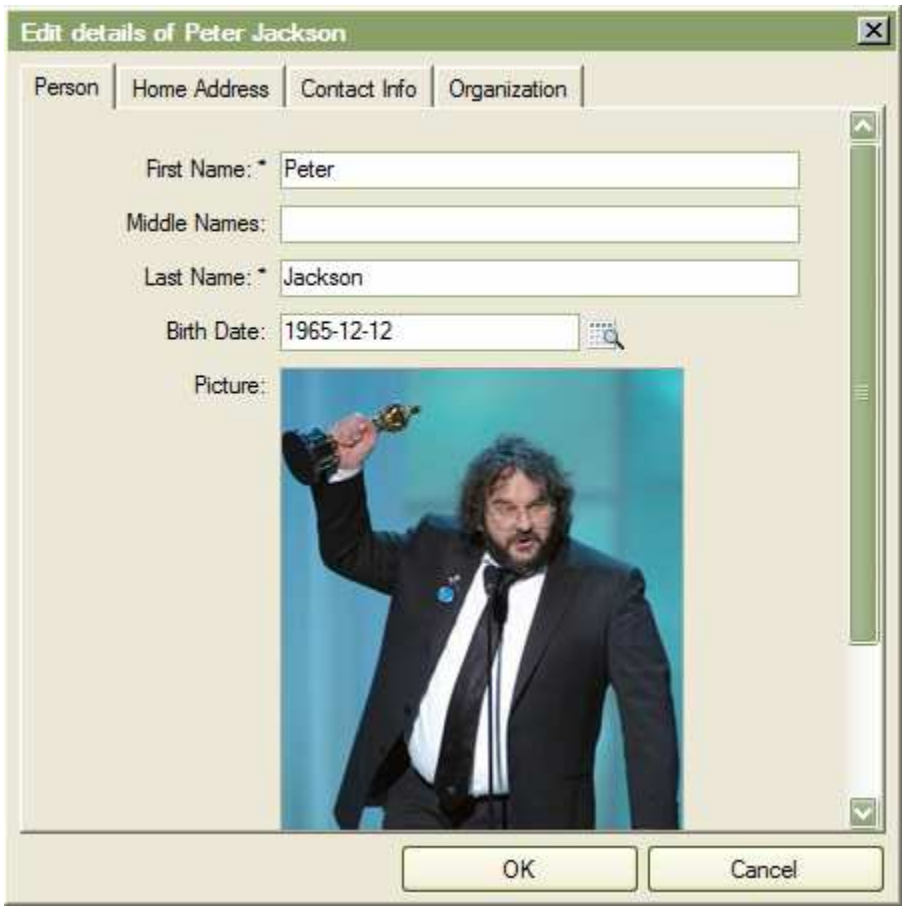
When renaming or removing a group, the tree control does not need to construct any new nodes, so the change is saved on the server without requesting any new nodes. The structural changes are processed client side resulting in a very fast update of the GUI.



Creating a new contact group

Creating, editing and deleting Contacts

Doing CRUD (Create, Update, Delete) operations on items in a paged dataset puts certain demands on the way the client handles changes. It is easy to forget that cached pages might no longer contain valid data after a change has been made on a different page. It is important to realize when client side data must be marked dirty (invalid). At the same time big savings can be made by being specific and not marking all data in a control as dirty when a change does not require it. Often a structural change can be handled on the client without repopulating the paged control completely.



Edit contact details in a modal dialog

The basic order of task execution when someone edited a contact's information and clicks the "OK" button is as follows:

- Some client side form validation is performed to ensure that the required fields are filled
- If validation failed, the user is notified and nothing else happens
- If validation was successful, the action attribute of the form is modified to ensure the data is saved in the right context
- The form is submitted
- All pages of the paged data control are marked dirty. This is done because to the client it is unknown on which page the new or edited contact will end up. Placement within a page is determined by the way the data is sorted, and sorting is done on the server because only the server has access to the complete dataset.
- Now that all pages have been marked dirty, the data control can be triggered to refresh the current page, and it won't be taken from cache
- Finally, the GUI gets a cleanup to prepare it for new input from the user. What exactly needs to be cleaned depends on which view is currently active: the list view or the card view

The code for saving and deleting contacts branches quite a lot, but it's actually not that complex. The reason why it branches like this is because there is often different code for the list view and the card view, and some optimizations have been made to avoid making needless server roundtrips. Most of this section of code has been commented, so it should be quite readable.

Conclusion

Hopefully this starter kit will come in handy if you are creating an application that makes heavy use of Backbase controls and data binding. I have hardly even mentioned nice hidden features like layout customization by resizing panels, dragging contact cards to the tree to move them to different groups, and uploading pictures to add them to contacts. Those are things you can investigate yourself and experiment with.

There are also endless more opportunities for extending this starter kit, for example:

- Remembering the user selected sort field and sort order with cookies
- Allowing the user to customize the number of contacts per page
- Importing files with contact information that were exported from other applications
- Using context menus to give an alternative way to access the editing features
- Adding email or calendaring functionality

Installation Guide

System Requirements

The Backbase Personal Information Manager Starter Kit requires the following:

- Any operating system
- PHP 4 or 5
- MySQL*
- Backbase Community Edition or Backbase Client Edition 3.1.4+

* You need a database for this starter kit: although we recommend MySQL it's also possible to use an other database system like MSSQL or Oracle. This is due to the fact that the starter kit is using an Abstract Database Object to communicate with any SQL server. For this guide we assume you use MySQL.

PHP & MySQL

First make sure you have a working PHP/MySQL installation: we assume you are familiar with this procedure, or that you can find all necessary information in the documentation of these products.

Database creation and configuration

Create a MySQL Database by executing the personmanagement.sql file within your installation. This can be done command-line or for example with a program like phpMyAdmin.

Configure database user access

After you have setup the database including permissions, you might need to edit the libs/datamanager.php file where the credentials for the user, password, database and host are stored.

Test!

Once these things are set up, you can run the Backbase Personal Information Manager Starter Kit.

If you have questions or installation problems, please visit our Examples and Documentation forum on the DevNet: http://www.backbase.com/#dev/home.xml