
Chapter 1. Writing Custom Xataface Actions

Table of Contents

What is an action?	1
Hello World	1
Using Templates	2
Hello World With Templates	2
Extending Existing Templates	3
What Slots are Available to Override?	5
Which Templates Can Be Extended?	7
Using a Custom Stylesheet	9
Using the custom_stylesheets slot	10
Using the Dataface_Application::addHeadContent()	12
Using Database Data	12
Displaying The Current Found Set	14
Permissions	16
Using checkPermission()	16
Record-Dependent Permissions	19
Custom Permissions	22
Using getPermissions()	26
Using the permission directive	26
Adding Buttons, Links, and Menus to the UI	26
The actions.ini file	26
Adding a Hello Button	28
Using PHP Expressions In Action Directives	38
"Selected Record" Actions	42
Creating a "Selected Records" Action	43
Building Selected Records Request Manually	47

What is an action?

In Xataface, an action is a named HTTP request handler. When you point your web browser at a Xataface application, Xataface will look at the `-action` request parameter to determine which action should be used to handle the request. An action can be defined by way of a PHP class located in the `actions` directory of your application, and it can be configured by adding a section to the application's `actions.ini` file. Configuration options include such things as the permission required to access the action, whether (and where) it should be accessed within the application menu systems, and which template should be used to render the action.

Hello World

For our first custom action, let's create a simple page that simply says "Hello World!!!". The following steps assume that you already have a working Xataface application located at `/var/www/myapp`, and web accessible at `http://yourdomain.com/myapp/`.

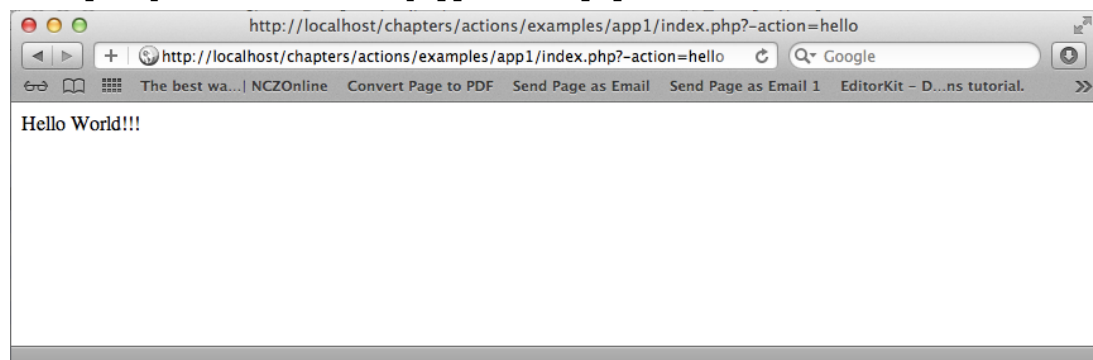
1. Create a directory named `actions` inside your application directory (i.e. `/var/www/myapp/actions`). This is where you will store all of your custom actions.

2. Create a text file inside the `actions` directory, named `hello.php`, with the following contents:

```
<?php
class actions_hello {
    function handle($params){
        echo "Hello World!!!";
    }
}
```

A couple of things to note about this class. The class name follows the required naming convention for action classes: `actions_<name>` where `<name>` is the name of the action. This must match the naming convention also for the containing file. I.e. If the action was called `goodbye`, then the class would be named `actions_goodbye`, and it would be contained in a file named `goodbye.php`.

3. Now, point your web browser to your application, with the `-action` parameter set to "hello". I.e. Go to `http://yourdomain.com/myapp/index.php?-action=hello`



Using Templates

One of the most powerful aspects of Xataface is its use of templates to separate the presentation from the business logic of the application. Xataface comes with the Smarty template engine [<http://smarty.net>] built in. Using templates to render content to the screen is preferred over using `echo` or `print` statements because it separates the visual aspects from the programming aspects to a large degree. A template is basically just an HTML page with some special macros and directives that can be used to embed content and provide some minimal logic.

In Xataface, all templates are stored in your application's `templates` directory. In order to improve performance, templates are compiled automatically by Xataface into PHP files that can be executed very quickly at runtime. If you have a working Xataface application, you have already created a `templates_c` directory to store these compiled templates. Xataface actually comes with its own set of built-in templates that are stored in the `xataface/Dataface/templates` directory. You should not modify any of these templates, but you can override any of them adding a template of the same name into your application's `templates` directory. If there are templates of the same name in both your application's `templates` directory and Xataface's `templates` directory, then your application's template will be used instead.

Xataface also allows you to extend templates by way of the `{use_macro}` and `{fill_slot}` tags. These extremely useful tags allow you to easily build your own templates that inherit the look and feel of any other section of the application.

Hello World With Templates

Let's expand on the earlier Hello World example, by modifying it to use templates.

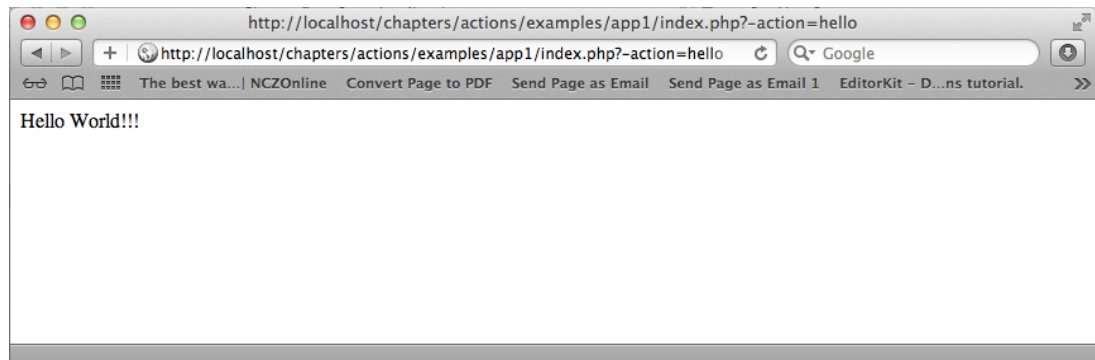
1. Create a templates directory in your application. (i.e. /var/www/myapp/templates).
2. Create a file inside your templates directory named hello.html, with the following contents:

```
Hello World!!!
```

3. Modify the actions/hello.php file so that it displays the hello.html template:

```
<?php
class actions_hello {
    function handle($params){
        df_display(array(), 'hello.html');
    }
}
```

4. Now, point your web browser to your application, with the -action parameter set to "hello". I.e. Go to `http://yourdomain.com/myapp/index.php?-action=hello`



Extending Existing Templates

So far, our hello action is uninspiring. It doesn't even maintain the look and feel of the rest of the application. Generally, you want your custom actions to fit into the look and feel of the rest of the site. Luckily this is easy to remedy using Xataface's `{use_macro}` and `{fill_slot}` tags. The `{use_macro}` tag allows you to "inherit" all of the content from another template. The `{fill_slot}` tag, that is placed inside the `{use_macro}` tag, allows you to override specific sections of the parent template.

The `Dataface_Main_Template.html` template (located in `xataface/Dataface/templates`) is the most frequently extended template because it provides application's entire look and feel (e.g. headers, footers, stylesheets, scripts, etc..). There are a number of slots that you can override, but the most common ones are as follows:

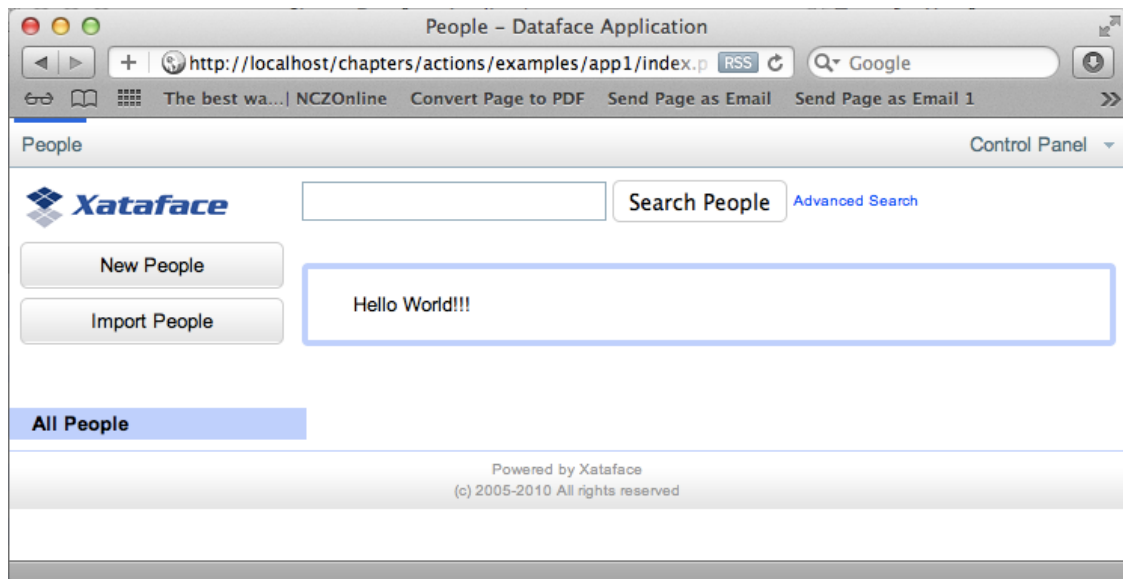
- `main_section` - The main content section underneath the search box in the middle of the page (spanning all the way to the right page margin).
- `main_column` - The entire main column. This engulfs and `main_section` slot and the search box, and any notices and messages that are posted. Generally you won't override both `main_column` and `main_section` at the same time. If you do override the `main_column` slot, you will need to provide your own display of error messages, since it will override the default error message sections.
- `html_body` - The entire HTML body. This will override all of the headers and footers, but will retain the document `<head>` so that the stylesheets and scripts will still be loaded.

Now, let's modify the `templates/hello.html` file to extend the `Dataface_Main_Template.html` template, and override the `main_section` slot so that our "Hello World!!!" text is displayed in the same look & feel as the rest of the application:

```
{use_macro file="Dataface_Main_Template.html"}
  {fill_slot name="main_section"}
    Hello World!!!
  {/fill_slot}
{/use_macro}
```

Now, reload your hello action. It should now look like Figure 1.1, "Hello World with Main Template (filling `main_section` slot)", with the proper Xataface header and footer. If you get a blank screen or an error message, then you probably have a syntax error. See if you can track it down with the error log. If you do see the Xataface header and footer, but don't see your "Hello World!!!" text, then you likely have a typo in your `{fill_slot}` tag. Check that you have the name correct.

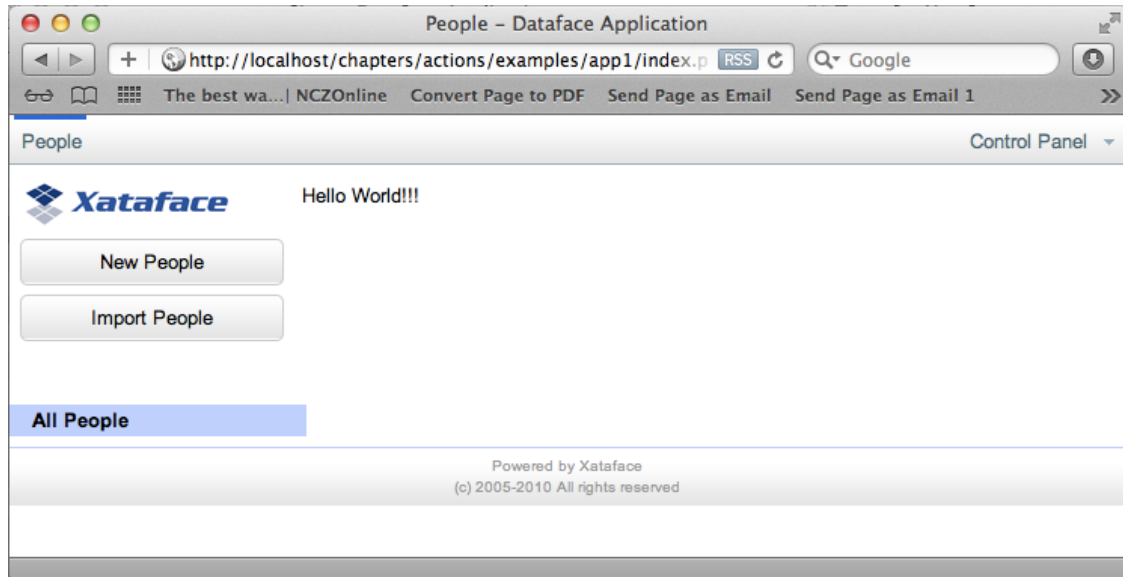
Figure 1.1. Hello World with Main Template (filling `main_section` slot)



Let's experiment a little bit with different slots. Figure 1.1, "Hello World with Main Template (filling `main_section` slot)" showed the result when we override the `main_section` slot. Now let's modify the template to override the `main_column` slot:

```
{use_macro file="Dataface_Main_Template.html"}
  {fill_slot name="main_column"}
    Hello World!!!
  {/fill_slot}
{/use_macro}
```

The result is shown in Figure 1.2, "Overriding the `main_column` slot".

Figure 1.2. Overriding the main_column slot

Notice the subtle change in the rendering of the hello action between Figure 1.1, “Hello World with Main Template (filling main_section slot)” and ????. When we override the `main_column` slot, it takes over the whole right column, eliminating the search box, and the blue border around the content.

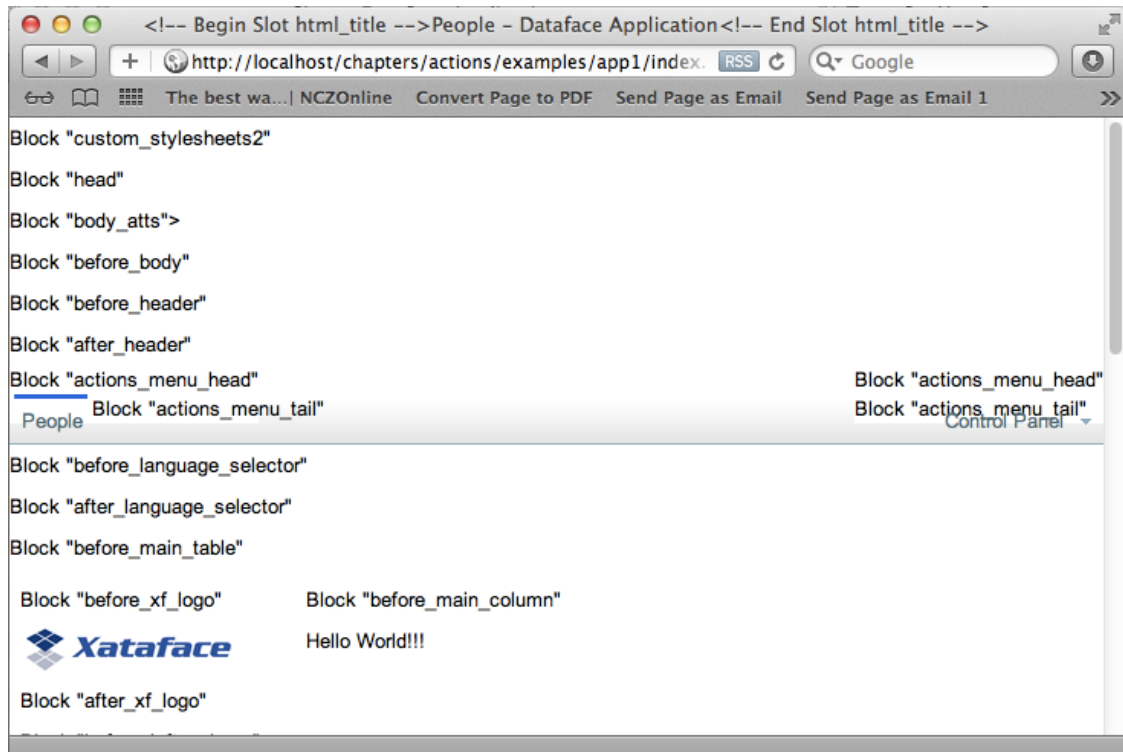
What Slots are Available to Override?

I have only mentioned three slots so far that can be overridden. However there are many different slots that can be overridden to place content just about anywhere you want to within the existing templates. If you want to see what slots are available to override, you can either open the template directly and look for `{define_slot}` tags, or you can turn on Xataface's debug mode and just reload the page. This will cause Xataface to print the names of all slots used as part of the page.

E.g., Add the following to the beginning of your application's `conf.ini` file:

```
debug=1
```

When you reload the page, you should see something like:



Notice all of the Block "before" strings. These indicate where there are blocks defined. Blocks are very similar to slots, except that they can only be overridden by methods in the delegate classes. If you want to see the slots that are available, you'll need to view the HTML source of this webpage, and look for `<!-- Begin Slot slotname -->` and `<!-- End Slot slotname -->` HTML comment tags.

Here is a truncated excerpt from that HTML source:

```
<!doctype html>
<!-- Begin Slot html_tag --><html lang="en"><!-- End Slot html_tag -->
<head>

<!-- Begin Slot html_head -->
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
  <title>
    <!-- Begin Slot html_title -->
      People - Dataface Application
    <!-- End Slot html_title -->
  </title>

  <!-- Begin Slot custom_stylesheets -->
    <!-- Stylesheets go here -->
  <!-- End Slot custom_stylesheets -->

  .... etc ....

<!-- End Slot html_head -->

</head>
```

You can see here, that there is a slot named `html_title`, that wraps the title of the page. Therefore, we can override this slot in our template to give it a custom title if we choose. Adding this to our template, we get:

```
{use_macro file="Dataface_Main_Template.html"}
  {fill_slot name="html_title"}Hello World!!!{/fill_slot}
  {fill_slot name="main_column"}
    Hello World!!!
  {/fill_slot}
{/use_macro}
```

And this will cause the `<title>` tags to now be rendered with the text "Hello World!!!".

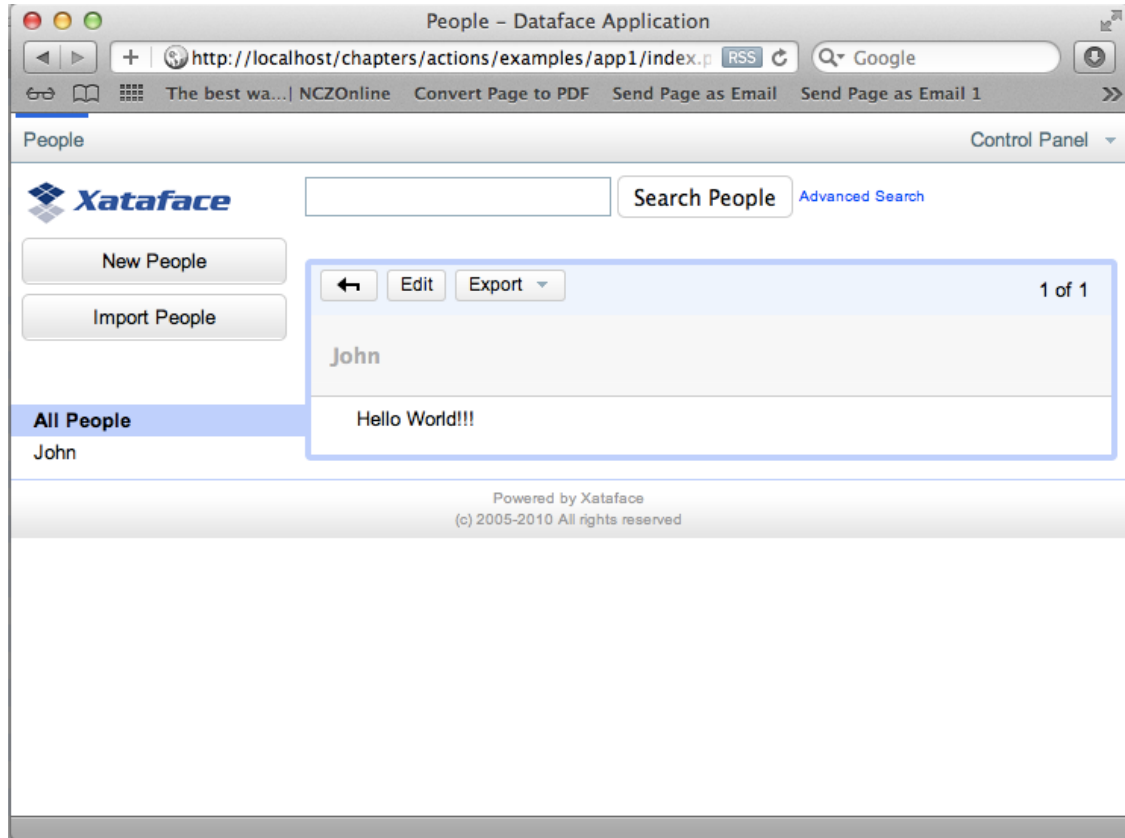
Which Templates Can Be Extended?

The previous examples all use the `Dataface_Main_Template.html` template as a base, however you can override any of Xataface's templates using the `{use_macro}` tag. In fact, depending on the context of your actions, you may be better to override a more specific template. The `Dataface_Record_Template.html` template provides the wrapper for the details view of a particular record. You can override the `record_content` slot to have your text displayed inside the record tabs (i.e. View, History, etc..).

As an example, let's modify the `hello.html` template to extend the `Dataface_Record_Template.html` template instead, and override the `record_content` slot.

```
{use_macro file="Dataface_Record_Template.html"}
  {fill_slot name="record_content"}
    Hello World!!!
  {/fill_slot}
{/use_macro}
```

The result follows:



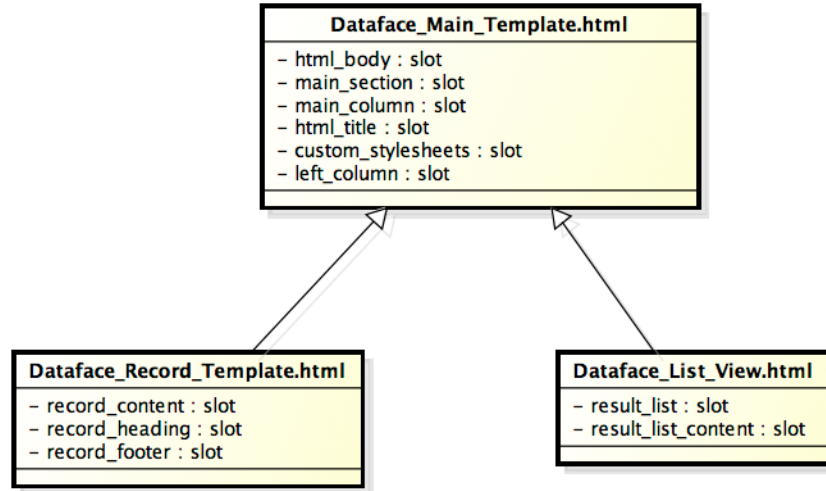
Another template that is useful to extend is the `Dataface_List_View.html` template. It includes two key slots:

1. `result_list` - Overrides the entire result list, including the controller buttons (i.e. next, previous, etc...).
2. `result_list_content` - Overrides only the result list, but leave the controller buttons in place.

For a comprehensive list of templates, you should browse the `xataface/Dataface/templates` directory and look through the files to see what templates are there. You can extend any of them. The main, three templates that are most common to override, however are:

1. `Dataface_Main_Template.html`
2. `Dataface_Record_Template.html`
3. `Dataface_List_View.html`

Figure 1.3, “Three commonly extended templates, and some of their key slots” shows the hierarchy of these 3 templates as a class diagram:

Figure 1.3. Three commonly extended templates, and some of their key slots

Using a Custom Stylesheet

There are a few different ways to include a custom stylesheet with your action. The most common methods are:

1. Implement one of the slots in the <head> of the document in your template, and include the stylesheet with a <link> tag. This method will work fine, most of the time, but it may conflict other blocks if you are trying to implement the same slot programmatically elsewhere inside the application.
2. Use the `Dataface_Application::addHeadContent()` method to add a <link> tag for your stylesheet. This method is the preferred method as it will always work, and should never conflict with other parts of the application.
3. Use the `//require-css` Javascript compiler directive in one of your Javascript files. This method is handy if the stylesheet is directly related to the content that is created or manipulated in the Javascript file, however it is less ideal than method #2 because it results in the stylesheet being loaded at the end of the page instead of the beginning. This can cause a split-second delay after page load before the stylesheet is applied to the page.

Regardless of which method you use to include your stylesheet, you will still need to create the stylesheet to begin with. In Xataface, it is a convention to save all of the stylesheets in the application's css directory (i.e. `/path/to/app/css`). So inside your application's directory, create the directory "css" if it doesn't already exist:

```
$ cd /path/to/app
$ mkdir css
```

Tip

You can save your stylesheets wherever you like; they don't have to be located in the css directory. However the css directory is added to the include path by default for the `CSSTool` class (and thus for the `//require-css` Javascript directive). What this means is that if your Javascript file contains the directive:

```
//require-css <actions/hello/hello.css>
```

then Xataface will try to load the file `/path/to/app/css/actions/hello/hello.css`. Since you will likely be creating the `css` directory and storing stylesheets in there for use by your Javascript libraries, why not just store all of your stylesheets under this path?

Before we create the stylesheet for our hello action, let's modify the HTML markup in the template so that it is easier for us to refer to it from CSS. We'll wrap the `Hello World!!!` text inside a `div` tag as follows:

```
{use_macro file="Dataface_Record_Template.html"}
  {fill_slot name="record_content"}
    <div id="hello-wrapper">Hello World!!!</div>
  {/fill_slot}
{/use_macro}
```

Now let's create a stylesheet that changes the font size and color of the text. Inside the `css` directory we created previously, create a new file named `"hello.css"`, with the following contents:

```
#hello-wrapper {
  color: blue;
  font-size: 24pt;
  text-decoration: underline;
}
```

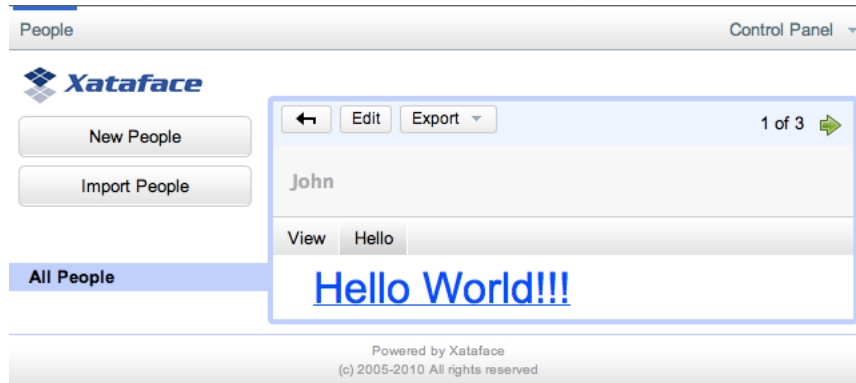
In the next two subsections we will use two alternative approaches for including this stylesheet in our action.

Using the custom_stylesheets slot

The first method we will try for including our stylesheet is by implementing the `custom_stylesheets` slot inside our template. Any content of this slot will be rendered inside the `<head>` tag when the page is published. Let's change our template as follows:

```
{use_macro file="Dataface_Record_Template.html"}
  {fill_slot name="custom_stylesheets"}
    <link rel="stylesheet"
        type="text/css"
        href="{ $ENV.DATAFACE_SITE_URL }/css/hello.css"
    />
  {/fill_slot}
  {fill_slot name="record_content"}
    <div id="hello-wrapper">Hello World!!!</div>
  {/fill_slot}
{/use_macro}
```

Now, if you reload your application and point it at the hello action you should see something like the following:



If you look at the HTML source for this page to try to find where the `<link>` tag was included, you'll see something like:

```
<!doctype html>
<html lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
    <title>People - Dataface Application</title>
    <link rel="stylesheet"
          type="text/css"
          href="/chapters/actions/examples/app1/css/hello.css"
    />
```

As you can see, this is included near the very beginning of the `<head>` section of the document. If you scan down a little further you'll notice that the rest of the Xataface stylesheets are actually included in a later part of the `<head>` section. This means that the default Xataface stylesheets will take precedence over the `hello.css` stylesheet in the case of an overlap or conflict. If this is an issue for you, you could opt to use the `head_slot` slot instead. A better solution, however, is to use the `Dataface_Application::addHeadContent()` method to add your stylesheet instead of using slots altogether.

Tip

An alternative way to fill slots is to implement a `block__slotname()` in either the table delegate class for a table, or in the application delegate class. We could achieve the equivalent inclusion of our stylesheet in the `custom_stylesheets` slot by implementing the method `block__custom_stylesheets()` inside the `people` table delegate class as follows:

```
function block__custom_stylesheets(){
  $app = Dataface_Application::getInstance();
  $query = $app->getQuery();
  if ( $query['-action'] == 'hello' ){
    echo '<link rel="stylesheet"
          type="text/css"
          href="'.DATAFACE_SITE_URL.'/css/hello.css"
        />';
  }
}
```

Using the Dataface_Application::addHeadContent()

It is quite common to want to add content into the <head> of the document output. Xataface provides a nifty method that can be called from anywhere in the application to add content into the <head>. Let's modify our action to use this method instead of slots.

First, let's remove the slot from the hello.html template to restore our template to normal:

```
{use_macro file="Dataface_Record_Template.html"}
  {fill_slot name="record_content"}
    <div id="hello-wrapper">Hello World!!!</div>
  {/fill_slot}
{/use_macro}
```

Next we'll add the following to the hello.php action handler any time before the call to df_display():

```
$app->addHeadContent(
    '<link rel="stylesheet"
      type="text/css"
      href="' . DATAFACE_SITE_URL . '/css/hello.css"
    />'
);
```

Now if you reload the hello action, it should look the same as before, with your custom styles applied to the content. If you look at the HTML source, you'll see that the stylesheet has been added in a different order than before, though.

Tip

You can also use addHeadContent() to include a global stylesheet (i.e. a stylesheet that is included everywhere in your application). The most common place to insert this call is in the beforeHandleRequest() method of the application delegate class. This method, as the name implies, is called before every page request. E.g., you might create a stylesheet called global.css, and then add the following to your application delegate class:

```
function beforeHandleRequest(){
    $app = Dataface_Application::getInstance();
    $app->addHeadContent(
        '<link rel="stylesheet"
          type="text/css"
          href="' . DATAFACE_SITE_URL . '/css/global.css"
        />'
    );
}
```

Using Database Data

None of the examples up to this point have made use of the database. Xataface includes a powerful database abstraction layer that allows you to load and save records from the database. It also provides methods to provide you with the user's current context within your app based on the Xataface URL conventions.

You can easily retrieve the "current" record if your action is meant to show details of a particular record. You can also retrieve the current found set if you want to show a list of records that match the user's current query.

Let's modify the `hello` action as follows:

1. Create a table in your database to store people:

```
create table people (  
    person_id int(11) not null auto_increment primary key,  
    first_name varchar(100),  
    last_name varchar(100),  
    email varchar(100)  
)
```

2. Modify the `hello.php` file as follows:

```
1 <?php  
    class actions_hello {  
        function handle($params){  
            $app = Dataface_Application::getInstance();  
5            $record = $app->getRecord();  
            if ( $record and $record->table()->tablename != 'people' ){  
                throw new Exception("This action only works on the people table."  
            }  
            df_display(array('person'=>$record), 'hello.html');  
10        }  
    }
```

We have made a few changes that are worth commenting on:

- a. On line 4, we obtain a reference to the `Dataface_Application` object. This is the grand central station of the Xataface application, providing a starting point for obtaining information about the current request.
- b. On line 5, we load the current record (as a `Dataface_Record` object) according to the user's current request parameters. This takes into account if the user has performed a search and then clicked on a particular record in the found set.
- c. Lines 6-8 check to make sure that the current record is indeed from the `people` table. The reason for this is that our template (that we will modify in the next step) will be assuming that it is displaying data from the `people` table. Some unexpected errors may occur if you allow users to access your action from a different table.
It is possible to write actions that will work on every table also. Most Xataface actions, in fact, work this way. In this example we limit to a single table for simplicity's sake.
- d. On line 9, we have modified the first parameter of the `df_display()` function so that the current record is passed to the template. In this case we have indicated that the record will be accessible in the template using the variable `$person`.

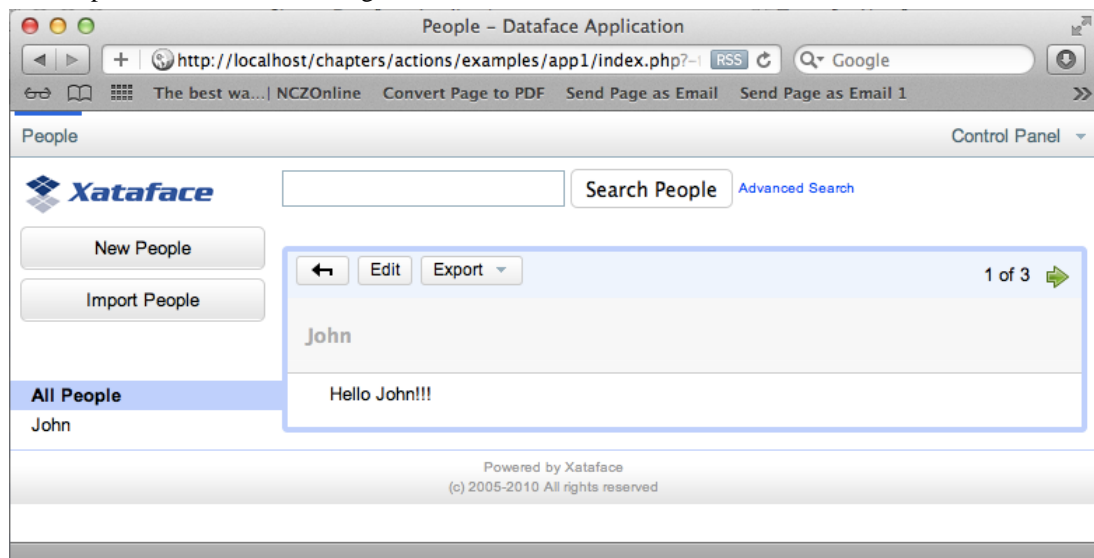
3. Modify the `hello.html` template as follows:

```
{use_macro file="Dataface_Record_Template.html"}  
    {fill_slot name="record_content"}  
        {if $person}  
            Hello {$person->htmlValue('first_name')}!!!  
        {else}  
            Hello to Nobody In Particular!!!  
        {/if}  
    {/fill_slot}
```

```
{ /use_macro }
```

There isn't very much new here. We are still extending the `Dataface_Record_Template.html` template so that our action content appears within the context of the current record. The only thing new is that we are displaying "Hello <name>!!!", where <name> is the first name of the current person record. Notice that we also deal with the case where no person was found. If the user performs a search and no people records are found, then the `$person` variable will be null. In this case, our action displays "Hello to Nobody In Particular!!!".

4. Open your application in your web browser and add a few people records using the "New People" form. Once you have added some people, enter the URL directly to your hello action: `http://yourdomain.com/myapp/index.php?-table=people&-action=hello`. Notice that you need to now include the `-table` parameter to make sure that we are currently in the people table. The output should look something like:



A few things to notice here:

- a. The current record's first name is "John". You can see this in the record heading where it displays the first `varchar` field by default - which is "John". Our content, "Hello John!!!", is correctly displayed in the record content section.
- b. If you click on the navigation arrow in the upper right (e.g. where it says "1 of 3") you can view your action for each of the people in the found set.

Displaying The Current Found Set

The previous example demonstrated how to obtain the current record. You may also want to provide a view that displays the current found set. The `Dataface_Application` class provides a `getResultSet()` method that will return the current results encapsulated in a `Dataface_QueryTool` object. Let's modify our `hello` action so that, instead of displaying a single record, it displays a list of all of the currently found records as a list.

We will achieve this by making the following changes:

1. Change the `hello.php` file as follows:

```
1 <?php
```

```

class actions_hello {
    function handle($params){
        $app = Dataface_Application::getInstance();
5        $results = $app->getResultSet();
        $results->loadSet();
        df_display(
            array('people'=>$results->getRecordsArray()),
            'hello.html'
10        );
    }
}

```

The key change here is on line 5 where we obtain the result set instead of the current record. We then pass this result set (on line 7) to the template in the `$people` variable.

2. Modify the `hello.html` template to extend the `Dataface_List_View.html` template and override the `result_list` slot:

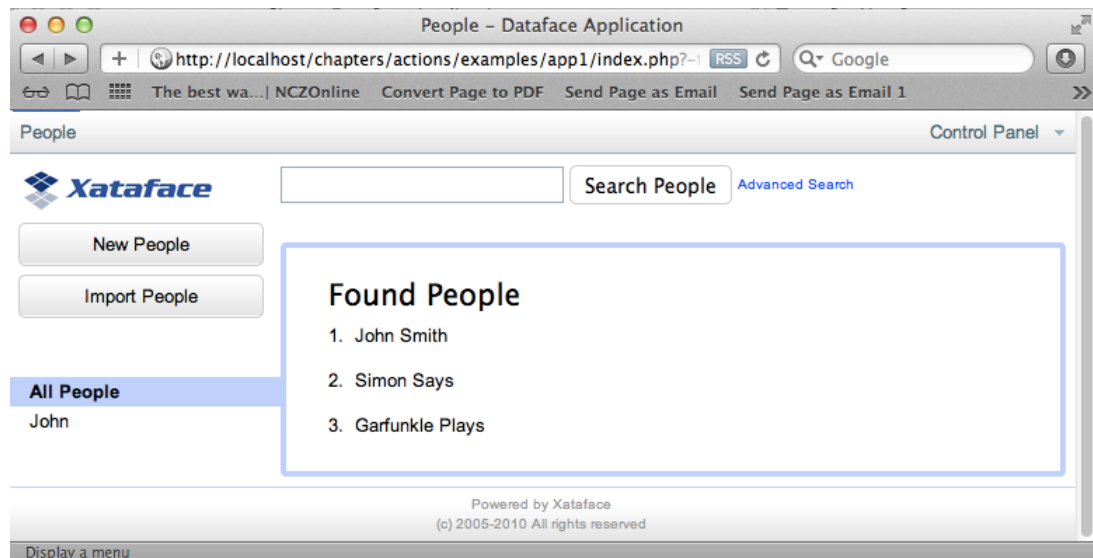
```

{use_macro file="Dataface_List_View.html"}
{fill_slot name="result_list"}
    <h1>Found People</h1>
    <ol>
        {foreach from=$people item=person}
            <li>{$person->htmlValue('first_name')}
                {$person->htmlValue('last_name')}
            </li>
        {/foreach}
    </ol>
{/fill_slot}
{/use_macro}

```

We make use of the `{foreach}` smarty tag in this example. It loops through each record in an array so that we can display a line item for each person in the found set.

3. Point your web browser to your hello action: `http://yourdomain.com/myapp/index.php?-table=people&-action=hello`. The output should look something like:



Permissions

By default, your custom action will not have any permission restrictions on it. This means that any user will be able to point their browser to your action and see some output. If you want to limit access to only certain users, then the best way is to use Xataface's built-in permissions infrastructure.

Xataface allows you to check the permissions that have been granted to the current user by way the following API methods:

- `Dataface_Record::checkPermission()`

Checks to see if a particular permission has been granted to the current user on a specific record.

- `Dataface_Record::getPermissions()`

Retrieves a permission-set that is granted to the current user on a specific record.

- `Dataface_Table::getPermissions()`

Retrieves a permission-set that is granted to the current user on records of a given table.

- `Dataface_RelatedRecord::checkPermission()`

Checks to see if a particular permission has been granted on the current user on a specific related record.

- `Dataface_RelatedRecord::getPermissions()`

Retrieves a permission-set that is granted to the current user for a specific related record.

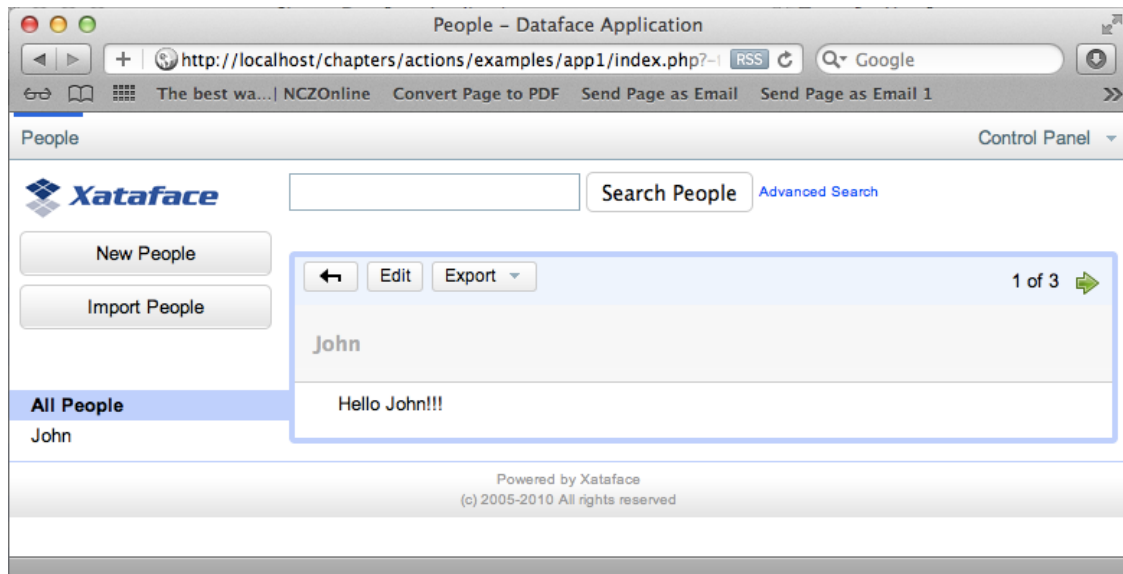
A common pattern for allowing/disallowing access to an action is to check to see if the user has been granted a certain permission, and then return a `Dataface_Error` object if the permission is denied. This will cause a "Permission Denied" error to be displayed to the user.

Using `checkPermission()`

```
<?php
class actions_hello {
    function handle($params){
        $app = Dataface_Application::getInstance();
        $record = $app->getRecord();
        if ( !$record->checkPermission('view') ){
            return Dataface_Error::permissionDenied(
                "You don't have permission to access 'hello'."
            );
        }
        df_display(array('person'=>$record), 'hello.html');
    }
}
```

In the above example, we load the current record and check to see if the current user has been granted the `view` permission. If this check fails, we call the `Dataface_Error::permissionDenied()` method that returns a `Dataface_Error` so that Xataface knows to either prompt the user to log in (if they aren't yet logged in) or to display a "Permission Denied" message.

If you haven't defined any permission rules for your application yet, then this won't actually make any difference. The output will be just the same as if you weren't doing a permissions check. E.g. If you point your browser to `index.php?action=hello`, you'll see something like the following:

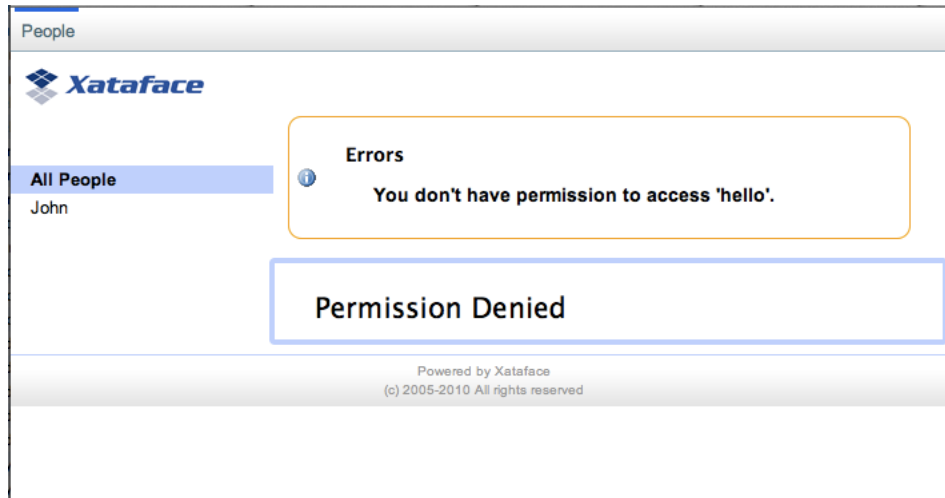


The reason for this is that, if you haven't specified any permission rules, then users are granted ALL permissions. In order to demonstrate the effect of adding this `checkPermission()` call in our `hello` action, let's add a permission rule to our application delegate class. The steps will be:

1. Create directory named "conf" inside your application directory (if it isn't already created).
2. Create a file named "ApplicationDelegate.php" inside this new conf directory with the following contents:

```
<?php
class conf_ApplicationDelegate {
    function getPermissions($record){
        return Dataface_PermissionsTool::NO_ACCESS();
    }
}
```

Now, if you try to access the hello action in your browser, you'll receive a "Permission Denied" message similar to the following screenshot:



This rule is overly restrictive and simplistic as it provides no access to all users, no matter who you are. Usually a `getPermissions()` method will include some control logic (i.e. if-else statements) to provide permissions to some users and not others. Our application doesn't currently have authentication set up, so it doesn't yet make sense to provide rules that discriminate against different users.

Let's take a moment to go through the flow of control so that we are clear on what is happening.

1. When Xataface receives a request for a URL with `-action=hello`, it will pass control to the `hello` action handler we created. Specifically, it will call its `handle()` method.
2. When it reaches the line with:

```
if ( !$record->checkPermission('view') ){
```

It asks Xataface whether the current user is granted permission to the 'view' permission on the specified record. Xataface will then ask the delegate class which permissions are granted to the current user by calling the `getPermissions()` method of the `ApplicationDelegate` class, passing it the `$record` as the context.

3. Our `getPermissions()` method in the delegate class returns `Dataface_PermissionsTool::NO_ACCESS()` which is a convenience method to return the ZERO permissions vector. I.e. no permissions are granted.
4. Xataface checks to see if this permissions vector grants the "view" permission. It does not, so Xataface will return `false` to the `$record->checkPermission()` call that was initiated in #2 above.
5. Our action returns `Dataface_Error::permissionDenied()` with the following code:

```
return Dataface_Error::permissionDenied(
    "You don't have permission to access 'hello'."
);
```

This will return a `Dataface_Error` object back to the Xataface dispatcher to let it know that permission has been denied to the action. What Xataface decides to do with this information depends on whether or not a user is currently logged in.

6. If a user is already logged in (or the application is not set up for authentication), Xataface will just display a "Permission Denied" message like the screenshot above. If, however, no user is yet logged in, and the application has authentication enabled, Xataface will redirect the user to a login form so that the user has an opportunity to provide credentials.

Record-Dependent Permissions

Our example from the previous section was a little boring, since the permission rule in the `ApplicationDelegate` class returned no permissions to anyone. It is more common, however, to return different permissions depending on the context (i.e. the record), the currently logged-in user, or both. Let's modify our permission rule so that users are granted ALL permissions on people records with first name "John", and no permissions on any other records.

This highlights a problem with our action. Our action is currently designed to work in the context of the people table, but there is nothing preventing the user from trying to access the action on any table. E.g. What if the user enters the URL: `index.php?-table=anothertable&-action=hello`? You'll receive a fatal error because the template will try to get the value of a column that doesn't exist in the current record. Therefore, it would be prudent to put in another check in our action to make sure that the current context is the people table. Our modified action will then look like:

```
<?php
class actions_hello {
    function handle($params){
        $app = Dataface_Application::getInstance();
        $query = $app->getQuery();
        if ( $query['-table'] != 'people' ){
            return Dataface_Error::permissionDenied(
                "The hello action only works on the 'people' table."
            );
        }
        $record = $app->getRecord();
        if ( !$record->checkPermission('view') ){
            return Dataface_Error::permissionDenied(
                "You don't have permission to access 'hello'."
            );
        }
        df_display(array('person'=>$record), 'hello.html');
    }
}
```

Now that we have locked our action into a single table, we can focus our permissions rules on the people table as well. In the previous example, we defined the permission rule inside the `ApplicationDelegate` class. It is strongly recommended that your `ApplicationDelegate` class defines very restrictive permissions, and then just override the permissions as required on a table-by-table basis. The recommended `getPermissions()` method for an application delegate class looks like the following:

```
<?php
class conf_ApplicationDelegate {
    function getPermissions($record){
        if ( isAdmin() ){
            return Dataface_PermissionsTool::ALL();
        } else {
            return Dataface_PermissionsTool::NO_ACCESS();
        }
    }
}
```

The `isAdmin()` function doesn't exist. It is a function that you would implement to return true only if the current user is the system administrator. The logic of this function is that only the system administrator can

do everything. All other users cannot do anything. This type of restrictive rule will help prevent accidental security mistakes, since you'll need to specifically grant access to users on a table-by-table basis.

In order to make our examples work, let's implement a dummy `isAdmin()` function by:

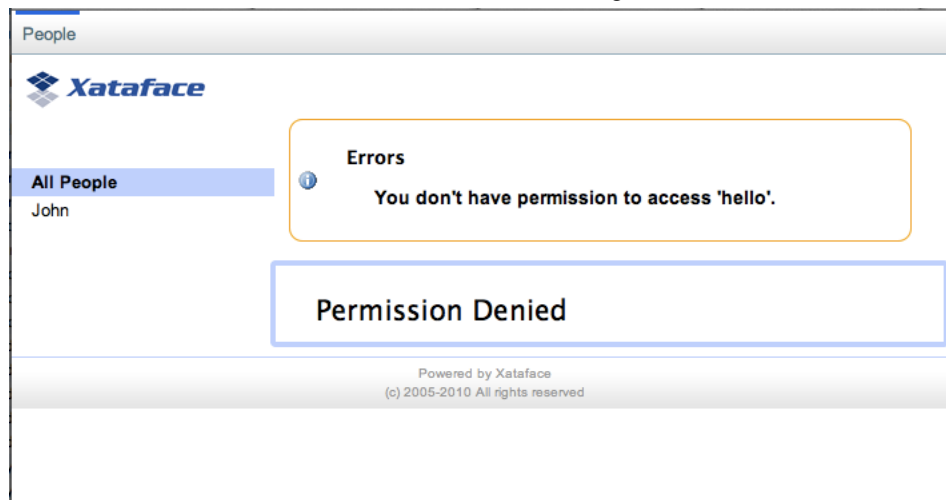
1. Create directory named "inc" inside the application directory.
2. Create a file inside the inc directory named `functions.inc.php` with the following contents:

```
<?php
function isAdmin(){
    return false;
}
```

3. Include this `functions.inc.php` at the beginning of your `index.php` file:

```
<?php
require_once 'inc/functions.inc.php';
require_once 'xataface/public-api.php';
df_init(__FILE__, 'xataface')->display();
```

At this point, you should try loading your `hello` action again just to make sure that it still works. All of these changes should have a "net-zero" effect on the application. This is because, even through we changed the `ApplicationDelegate` class to return permissions conditional on the result of the `isAdmin()` function, we defined the `isAdmin()` function to return false every time. Therefore, the output of the `hello` action should still be a "Permission Denied" message:



Finally, we can proceed forward with our initiative to allow access to the `hello` action only when viewing people records with `first_name` John. We'll do this by defining a permission rule inside the `people` table delegate class as follows:

1. Create the directory `tables/people` inside your application's directory if it doesn't already exist:

```
mkdir tables tables/people
```

2. Create a file named `people.php` inside the `people` directory that we just created. It should contain the following contents:

```
<?php
class tables_people {
```

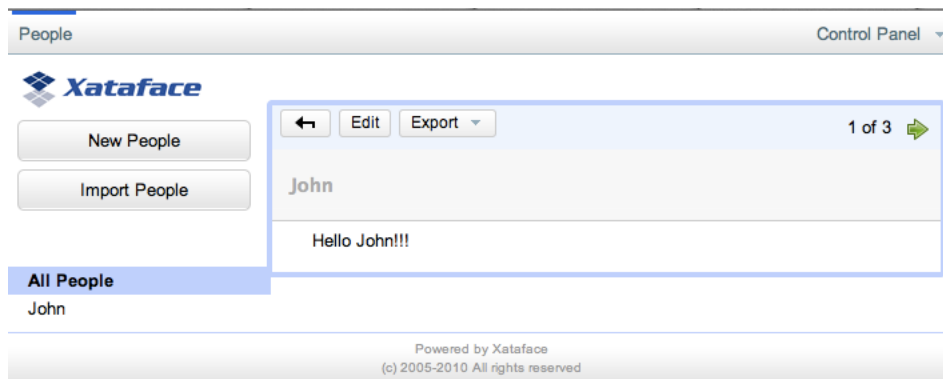
```

function getPermissions($record){
    if ( isAdmin() ) return null;
    if ( $record and $record->val("first_name") == "John" ){
        return Dataface_PermissionsTool::ALL();
    }
    return null;
}
}

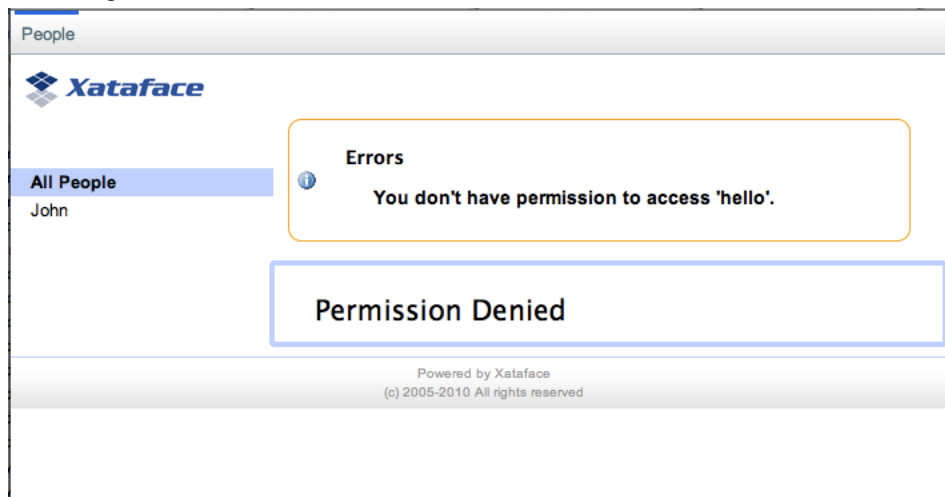
```

This class is the delegate class for the people table. It can be used to extend and customize the application in far more ways than just defining permissions rules. The `getPermissions()` method overrides the permissions defined in the `ApplicationDelegate` class such that records of the people table where the value of the `first_name` column is "John" are granted ALL access to users. Other situations aren't dealt with by this method. The call to return null causes the permission rules to be deferred to the `ApplicationDelegate` class rules.

Now, let's reload our hello action in the web browser and see if we are allowed to see its content.



You should be able to see the content for the first record, if your database data is the same as in this example (i.e. the first record has first name "John"). However if you try to view any other records, you will be disallowed. E.g. Click the "Next" button in the upper right to go to the next record, and you'll see something like:



Custom Permissions

The examples from the previous section used the "view" permission to restrict access to our hello action. While this may be appropriate, it ties the action to a permission that is used for other things as well. In some cases, you may want to insulate the hello action permission from other system permissions so that you can allow/disallow access to it independent of the user's access perform other actions. For example, you may want the user to have access to the hello action for a record, but not the view tab. If the hello action is locked down with the "view" permission, then this would not be possible.

In this section we will create our own custom "hello" permission that can be used to restrict access to the hello action. The only change to our existing code will be to change:

```
if ( !$record->checkPermission('view') ){  
  
to  
  
if ( !$record->checkPermission('hello') ){
```

If you make this change in your actions/hello.php file and then reload your hello action in the browser, you should notice that you don't have access to view any of the records anymore (not even the one with first name "John"). This is because the `Dataface_PermissionsTool::ALL()` method that we are using to return the set of allowed permissions for our user (in the case of a record with first name "John") only returns the set of "known" permissions in the system. Xataface doesn't know anything about the "hello" permission, so it is not included in the set of all permissions.

We need to define the hello permission in our application's `permissions.ini` file before we can effectively use it. The next two subsections give a brief overview of the `permissions.ini` file. For a complete coverage you should refer to the chapter on security.

The permissions.ini file

The `permissions.ini` file is a configuration file that defines all of the permissions and permission sets in your application. Xataface comes with a built-in `permissions.ini` file that defines the default permissions that are used by all of Xataface's internal actions. It is located in the xataface root directory. Like all INI files, the `permissions.ini` file consists of two scopes of configuration:

1. Global Scope - These are all the directives (`key=value`) that appear before the first section heading (i.e. `[sectionname]`).
2. Section Scope - These are all of the directives (`key=value`) that appear after one section heading (i.e. `[section1]`) and before the next section heading (i.e. `[section2]`).

All of the permissions are defined in the global scope of the `permissions.ini` file. All of the permission sets are defined as sections. The directives in each permission set either grant or deny a permission as part of that permission set.

Note

Permission Sets are sometimes referred to as "roles", since they can be used to define the set of permissions that are granted to a particular user role. The method `Dataface_PermissionTool::getRolePermissions($role)` actually returns a permission set defined by the given `$role`.

In order to get a feel for how the `permissions.ini` file works, let's take a look at the xataface `permissions.ini` file located at `/path/to/xataface/permissions.ini`. The beginning of it contains just simple permission declarations:

```
;;First we will list the permissions and their associated labels.
```

```
;; view : Allowed to view an element  
view = View
```

```
;; link : Allowed to access the link to a record  
link = Link
```

```
view in rss = View record as part of RSS feed
```

```
;; list : Allowed to see the list tab  
list = List
```

```
calendar = calendar
```

```
;; edit : Allowed to edit information in an element  
edit = Edit
```

```
;; new : Create a new record  
new = New
```

A few things to notice from this excerpt:

1. INI files use semi-colons for comments. I.e. everything after ";" on a line is considered a comment.
2. The left side of an equals sign specifies the name of a permission that is defined in the system. This is the value that will be referred to methods like `Dataface_Record::checkPermission($perm)`.
3. The right side of an equals sign is a human-readable label for the permission. This is not used in Xataface. It is just for your benefit as a programmer so that you can understand what a permission is for.

The second part of the `permissions.ini` file consists of permission-set definitions. They look like:

```
[READ BASIC]
```

```
;; A role that allows reading through the web but not in other structured formats  
;; like CSV, XML, RSS, or JSON
```

```
view = 1  
link = 1  
list = 1  
calendar = 1  
show all =1  
find = 1  
navigate = 1  
find_list =1  
find_multiple_table = 1  
view related records = 1  
find related records = 1  
link related records = 1  
expandable = 1
```

```
[READ ONLY]
```

```
view in rss=1  
view = 1  
link = 1
```

```

list = 1
calendar = 1
view xml = 1
show all = 1
find = 1
navigate = 1
ajax_load = 1
find_list = 1
find_multi_table = 1
rss = 1
export_csv = 1
export_xml = 1
export_json = 1
view related records=1
related records feed=1
expandable=1
find related records=1
link related records=1

;;-----
;; The EDIT role extends the READ ONLY role so that anyone who can edit can also
;; READ. It is pretty far reaching, as it provides permissions to edit records,
;; and manipulate the records' relationship by adding new and existing records
;; to the relationship.

[EDIT BASIC extends READ BASIC]
edit = 1
add new related record = 1
add existing related record = 1
add new record = 1
remove related record = 1
reorder_related_records = 1
import = 1
translate = 1
new = 1
ajax_save = 1
ajax_form = 1
history = 1
edit_history = 1
copy = 1
update_set = 1
update_selected=1
select_rows = 1
update related records = 1
edit related records = 1

```

This excerpt shows the definition of three permission sets:

1. READ BASIC

A collection of all of the permissions that allow a user to "read" content through the web. This is a convenient set of permissions to be granted to users if you want them to be able to view content for a record, but not edit it.

2. READ ONLY

A collection of all of the permissions that allow a user to "read" content. This includes all of the permissions of READ BASIC, and a few other permissions that pertain to access to web services and XML output.

3. EDIT BASIC

A collection of permissions that allow the user to read and edit content through the web. Notice that this definition uses the "extends" keyword to extend the READ BASIC permission set. This means that it includes all of the permissions defined in READ BASIC, but overrides them with the permissions it defines in itself.

A few observations from this excerpt relating to the format of permission sets:

1. Each directive in a permission set section specifies that a particular permission is either "granted" or "denied" as part of this permission set.
2. Assigning a value of "1" to a permission, indicates that that permission is "granted".
3. Assigning a value of "0" to a permission indicates that that permission is "denied".
4. The absence of a permission in a permission set means that the permission is "undefined" as far as this permission set goes. An undefined value would default to "denied", unless the permission-set is merged with another set that explicitly grants the permission. This might occur if you are granting a permission set to a user for a particular field in a record, and a different set for the parent record itself. When calculating the effective permission on the field it will merge the record permission set and the field permission set. If the permission is omitted in the field set but granted in the record set, then the effective permission would be "granted". This is a finer point that you don't need to understand for the purposes of this tutorial. For more information about this please refer to the security chapter.
5. The "extends" keyword can be used to inherit the permissions from another permissions set. This is very handy if you want to create a permission set that is the same as an existing set except for just a few aspects. Then you can just provide explicit permission directives for the differences.

The Application's permissions.ini file

Generally, you should not modify the `permissions.ini` file that is located in the xataface directory. Like with most things "xataface", you are encouraged to create your own `permissions.ini` file inside your application's directory which will extend and override the Xataface `permissions.ini` file. In this file, you can define your own permissions and permission sets. You can also extend and override permission sets that are defined in the Xataface `permissions.ini` file.

Defining the hello Permission

Now we return to our example in which we want to restrict access to the `hello` action based on whether the current user has been granted the `hello` permission. We have set up a rule in the `people` table's `getPermissions()` method to allow ALL permissions to the current user on records with first name "John". We just need to define the "hello" permission in our application's `permissions.ini` file in order for it to be included with the set of ALL permissions. We will proceed as follows:

1. Create a file named `permissions.ini` inside our application's main directory (if it doesn't already exist).
2. In the global scope (i.e. before the first section heading), add the following directive:

```
hello = "Permission to perform hello action"
```

Now try to reload the `hello` action, and you should again be able to access the record with first name "John".

Using `getPermissions()`

The previous section made use of the `Dataface_Record::checkPermission()` record to see if the current user is granted a particulate permission with respect to a record. An alternative way to check permissions is using the `getPermissions()` method on the `Dataface_Record` class. Whereas `checkPermissions()` returns a boolean value, `getPermissions()` will return an associative array of permission names mapped to a boolean value indicating whether the user has been granted that permission. We could write our previous example equivalently as:

```
<?php
class actions_hello {
    function handle($params){
        $app = Dataface_Application::getInstance();
        $record = $app->getRecord();
        $perms = $record->getPermissions();
        if ( !$perms['hello'] ){
            return Dataface_Error::permissionDenied(
                "You don't have permission to access 'hello'."
            );
        }
        df_display(array('person'=>$record), 'hello.html');
    }
}
```

Using the permission directive

Another way of limiting access to an action is via the `permission` directive in the `actions.ini` file. This method is discussed in [yyy](#).

Adding Buttons, Links, and Menus to the UI

So far, we've only discussed actions from the perspective of the back-end of the system. I.e. We know how to write the logic and control-flow to handle HTTP requests that are submitted to the application by a user. This is only half of the story, however. The other half involves how actions are embedded in the user interface. You may want to add a button or a menu-item somewhere in the user interface so that the user can access your action. This is where the `actions.ini` file comes into play.

The `actions.ini` file

The `actions.ini` file is a configuration file that can be added to the main directory of your application to specify how actions should be integrated into the user interface. It also allows you to configure some non-UI aspects of your actions such as permissions.

Each section of the `actions.ini` file corresponds with an action, or button/menu-item, or both. Some common directives that you can assign to an action include:

- `label` - Specifies the label for the action's menu item in the UI.
- `description` - Specifies the tool-tip text that is shown when you hover over the action's menu item.
- `url` - Specifies the URL link where the user is directed when they click on your action's menu item.

- `category` - The name of the category that this action belongs to. Among other things, this dictates where the action will be displayed in the UI. For example, Actions in the `record_actions` category are rendered as buttons along the top of the details panel for a record.
- `condition` - A PHP expression whose resulting boolean value dictates whether the action's menu-item will be displayed in a particular context. For example, you may want an action to be visible only for certain tables or only when certain other actions are currently being displayed.
- `permission` - The name of a permission that the user needs to be granted in order to see the action's menu-item or to access the action at all.

The core Xataface actions are defined in the Xataface `actions.ini` file inside the xataface directory. In your application's `actions.ini` file, you can define your own actions, or extend and override actions from the Xataface `actions.ini` file.

To get a feel for the anatomy of an `actions.ini` file, let's take a look at the Xataface `actions.ini` file (located at `path/to/xataface/actions.ini`). The following is an excerpt with definitions for the browse, list, and find actions:

```
;;-----
;; Table tabs
;; -----
;;
;; The table tabs are the little tabs ('details', 'list', 'find', ...) at the top
;; of the screen.

;; Show the details of the current record
[browse]
    label = Details
    category = table_tabs
    url = "{$this->url('-action=view')}"
    accessKey = "b"
    mode = browse
    permission = view
    order=0

;; Show a list of the records in the current found set
[list]
    label = List
    category = table_tabs
    url = "{$this->url('-action=list')}"
    accessKey = "l"
    mode = list
    template = Dataface_List_View.html
    permission = list
    order=0.5

;; Show a "Find Record Form"
[find]
    label = Find
    category = table_tabs
    url = "{$this->url('-action=find')}"
    accessKey = "f"
    mode = find
    permission = find
```

```
template = Dataface_Find_View.html
order=0.75
```

Some things to notice from this excerpt:

1. Section headers (i.e. [section]) correspond with action definitions.
2. Directives following a section header (but before the next header) are directives that pertain to that section's corresponding action.
3. Directives can contain PHP expressions that draw on a limited execution context. E.g. the `url` directives all use the `Dataface_Application::url()` method to dynamically generate the url for the action for the appropriate context.

Note

The `$this` variable in an action directive's PHP expression will always refer to the `Dataface_Application` object. For a full list of variables available in this execution context see xxx.

4. The list and find actions both include a `template` directive. This is an alternative to providing a PHP action handler. If you look in the `xataface/actions` directory, you won't see handlers for list or find. This is because these actions are configured to just render their respective templates directly without any special processing by an action handler.
5. All three of these actions have the same value for their `category` directive. This specifies that these actions should be displayed in the part of the UI where the `table_tabs` are rendered.

Note

Starting with version 2.0, Xataface comes standard with the `g2` module, which defines its own `actions.ini` file (and some other files) that extends the default Xataface one, and overrides some of the actions. The `g2` module changes the layout of the user interface, and does away with the familiar table tabs. It does this by overriding the find and browse actions in its `actions.ini` file as follows:

```
[browse > browse]
    category=" "

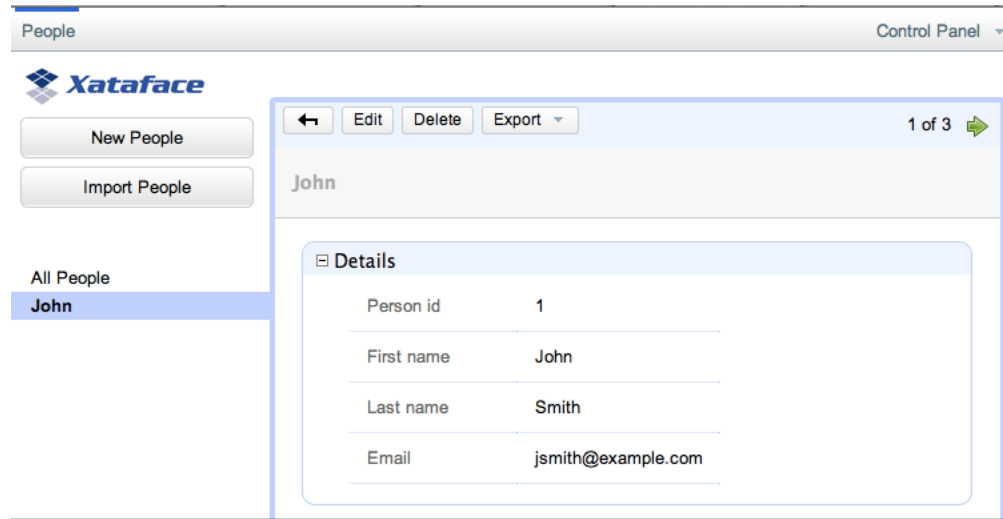
[find > find]
    category=" "
```

This effectively removes the category on these actions so that they will not appear in any category. If you were trying to find out where these actions are manifested in the UI and you have the `g2` module installed (the default on Xataface 2.0), then this is likely the reason why you cannot find them.

Adding a Hello Button

Now we are ready to add a button that links directly to our hello action. Since it pertains to a particular record, it makes sense for this button to appear in the context of a particular record. In order to make this happen, we need to know what category to assign to the action definition in the `actions.ini` file. A good strategy, when trying to choose a category, is to look through the user interface to find existing menu items and buttons that reside in the context and location where we want to add our button.

Let's take a look at the details view for a record, shown below:



A good place for our "Hello" button appears to be along the top of the details panel where the Edit, Delete, and Export buttons are displayed. If we can find out what category is assigned to these actions, we can assign the same category to our button and have it displayed along with them. Finding the category may involve some digging. My general procedure involves:

1. Open the Xataface actions.ini file and do a find for the labels of the actions in question. E.g. We could open the xataface/actions.ini file and search for "Edit" within the file. The fifth or sixth result will be the definition for the [edit] action:

```
;; Edit the details of the current record.
[edit]
    label = Edit
    url = "{$this->url('-action=edit&-relationship=')}"
    template = Dataface_Edit_Record.html
    mode = browse
    category = record_tabs
    selected_condition = "$query['-action'] == 'edit'"
    permission = edit
    order=-1
```

You can see that the category of this action is "record_tabs".

Note

The category displayed for an action in the Xataface actions.ini file may not be the effective category for the action. Modules can include their own actions.ini files that extend or override actions in Xataface. Your application may also override actions in its own actions.ini file. Xataface loads actions.ini file in the following order:

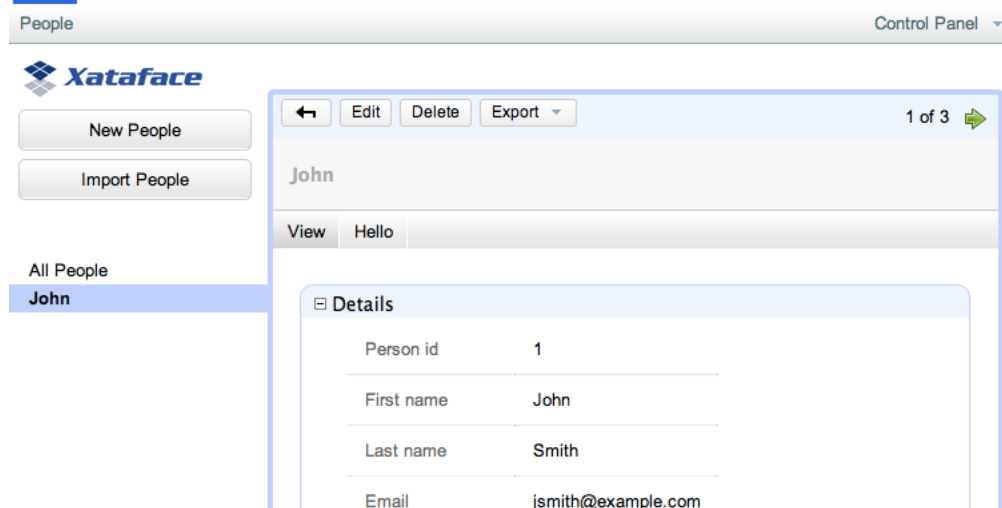
- a. xataface/actions.ini
- b. modules/*/actions.ini (in the order that they appear in the [_modules] section of the conf.ini file.
- c. The application's actions.ini file.

2. If you found a matching action in step 1, skip to step 4. Otherwise, proceed to step 3.

3. If you did not find a matching action in the `actions.ini` file for the button or menu item you are interested in, check your application's `conf.ini` file to see which modules are currently loaded. These modules will be listed in the `[_modules]` section. Check each of these modules to see if they contain an `actions.ini` file. Search in these files to see if the action you are looking for is defined.
4. If your application doesn't have its own `actions.ini` file yet, then create it. I.e. create an empty file inside your application's main directory named `"actions.ini"`.
5. Try creating a dummy action with the same category as the action that you found. In our case we would do something like:

```
[hello]
    category=record_tabs
```

If we reload the application and go to the details view for a record, we'll see that something unexpected has happened.



A new "Hello" tab has appeared below the record title. This is now where we expected it to show up though. Why is that?

The answer lies in the fact that Xataface 2.0 comes with the `g2` module installed by default. The `g2` module overrides many of the default Xataface templates and actions to reorganize the look and feel. The "Edit" action, in the default Xataface `actions.ini` file, is assigned the `"record_tabs"` category. But if you check the `xataface/modules/g2/actions.ini` file, you will see that the `edit` action has been overridden with the following configuration:

```
[edit > edit]
    category=" "
```

This was done to remove the old `edit` menu items from the interface altogether. If you look through the rest of the `g2 actions.ini` file, you'll notice that there are a few new action definitions that replace the old `edit` action:

```
[edit_record > edit]
    category=record_actions
    order=-100
    description="Edit this record"
    condition="$query['-action'] != 'edit'"
```

```
[cancel_edit_record > view]
  label="Cancel"
  description="Cancel Edit"
  condition="$query['-action'] == 'edit'"
  category=record_actions
  order=-100
```

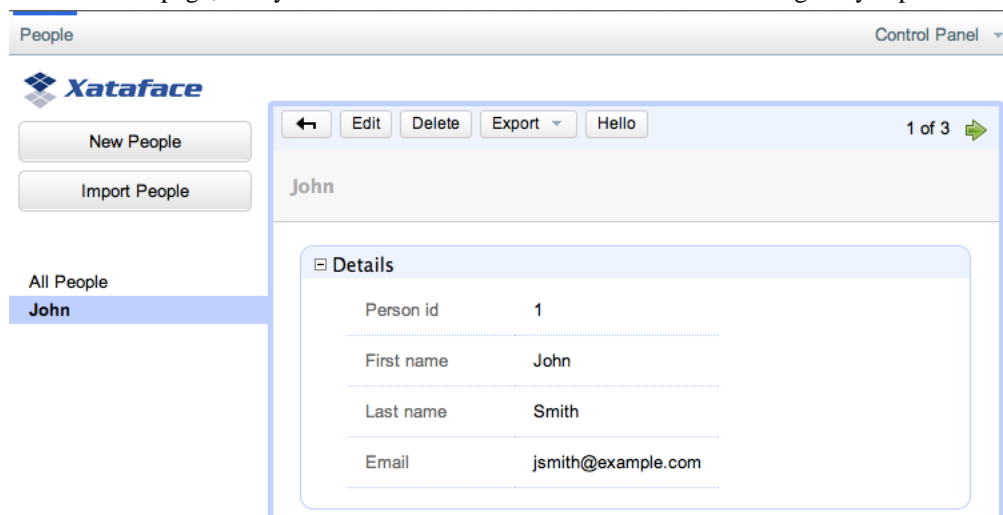
The edit action was extended by the new `edit_record` action, which is assigned the `record_actions` category. This action, however, is programmed to not appear in the edit form itself (the condition directive ensures this). The `cancel_edit_record` action is then defined to appear only on the edit form as a means of backing out of the edit form. These changes were made to improve the usability of the application. In Xataface 1.x, the edit form was accessed via a tab. Users, however, have come accustomed to using buttons to perform actions like editing a record. This is why the `g2` module removes the original action and replaces it with a different one in a different location.

This also demonstrates the flexibility of Xataface's *actions* architecture. You can override and modify any action in the system without having to modify the core source files.

If you change the category of the "hello" action to "record_actions", as follows:

```
[hello]
  category=record_actions
```

and reload the page, then you should now see a "Hello" button where we originally expected it to appear:



After all this, it turns out that the `record_tabs` category actually is a more appropriate place for our `hello` action. The `record_actions` buttons look like they are all functions that can be performed on the record. Our `hello` action, however, just displays some information about the record, and this is probably more appropriate as a tab.

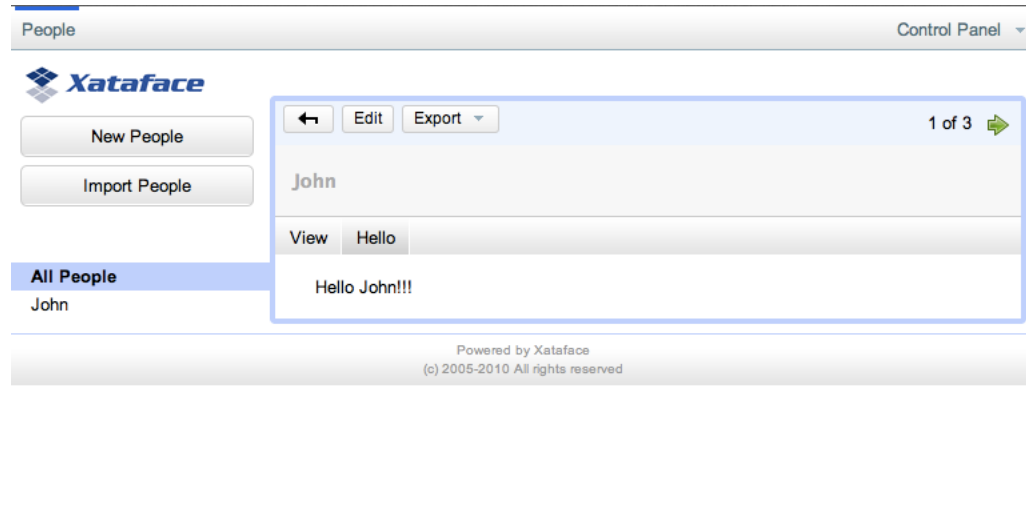
Handling Button/Menu Click

Once we have changed it back to the `record_tabs` category, we can proceed to make the button useful. Currently, when you click on the "Hello" button, it doesn't do anything. This is because we haven't specified the `url` directive yet. We will do that now:

```
[hello]
  category=record_tabs
```

```
url="{ $this->url( '-action=hello' ) }"
```

What we are doing here is calling the `url()` method on the `Dataface_Application` object. This method generates a URL that maintains the same context as the current page, except that it is overridden by the parameters it receives. In this case, the link will be to the same URL as the current page (so it will be pointing at the same result set and the same record), except that the `-action` is changed to "hello". If you click on this button now, it will take you to our hello action:



Note

Using `{ $this->url('-action=myaction') }` in the `url` directive is one of many ways to add functionality to a button. Other ways include, `{ $record->getURL('-action=myaction') }` which would generate a URL to the context record rather than maintaining the current result set context. E.g.:

```
[hello]
  category=record_tabs
  url="{ $record->getURL( '-action=hello' ) }"
  url_condition="$record"
```

In this case you need to also include the `url_condition` directive which tells xataface to only evaluate the URL if the `$record` variable is non-null. (Without this, if you have an empty found set, a fatal error would occur because Xataface would try to call `$record->getURL()`, but `$record` would be null.

If your action could appear in contexts other than the details page of a record, then you should use this alternate `$record` approach for defining your links, since it guarantees that the link will always go to the intended record. The `$this->url()` approach will merely generate a link in the same context as current.

An alternative way to handle clicks is using Javascript. The most common way to accommodate this is to add the `class` directive to the action, which will manifest itself as a CSS class for the `` tag in the resulting HTML page. Then you can use this CSS class to select the button using jQuery and add a handler in one of your Javascript files. See section xxx for more coverage on this strategy.

Attaching Javascript Event Handlers

In modern applications, users are coming to expect more interactivity from their applications. If you require more flexibility on a button/menu link than simply linking to another page, then you'll probably need to use Javascript in some way to handle the "click" event on the button. There are many ways to do this, but the recommended method involves adding the class directive to your action definition (which adds a CSS class to the tag that wraps the button or menu item), then using Javascript to retrieve a reference to the button based on this class. The part that may require some thought is how best to include your Javascript file in the page.

Including Javascript files in your application is very similar to including CSS files. In fact any of the strategies mentioned previously for including CSS files can be used for including Javascript files. The tricky part is deciding when you need to include your javascript file, and when it is okay to leave it out. If your Javascript includes a handler for the "hello" button, then you can't just simply include it in the hello action template. This is because the button appears on many pages - not just the hello action. You could decide to include your script on every page, but then you might be contributing to more bloat than necessary. Usually it is preferable to only require the client browser to download the resources that it needs. Therefore, the key to optimal placement is to find a way to include your script whenever the hello button appears, and not include it otherwise.

One strategy is to identify a block or slot in a template that is always present when the hello button appears. Since we have assigned the hello button to the record_tabs category, it will appear in any page that the record details appear. Therefore, we could include the <script> tag in one of the slots in the Dataface_Record_Template.html template. To see the available slots, you can either open the Dataface_Record_Template.html template, or enable debug in Xataface. Some of the candidate slots include:

1. before_record_content
2. after_record_content

For our action, we'll insert our script in the before_record_content block as follows:

1. Implement a method named block__before_record_content() inside your people delegate class (i.e. in tables/people/people.php):

```
function block__before_record_content(){
    echo '<script src="'.DATAFACE_SITE_URL.'/js/hello.js"></script>';
}
```

Tip

Although this example simply echos a <script> tag, the preferred way to include Javascripts is using the Dataface_JavascriptTool class. It supports dependency inclusion, minification, and also merges all of the scripts into a single file for reduced network latency. For a full discussion of on the Javascript tool see section xxx, but for now, consider the following snippet that is the equivalent to the above example:

```
function block__before_record_content(){
    $jt = Dataface_JavascriptTool::getInstance();
    $jt->import('hello.js');
}
```

Notice that, in this case, we don't need to include the 'js' directory in the path to hello.js. This is because the Javascript tool includes the application's js directory in its include path by default.

2. Add a "class" directive to the hello action definition in the `actions.ini` file:

```
[hello]
  category=record_tabs
  url="#"
  class="hello-action"
```

Note

We also changed the url directive to "#", which is basically a safe way of saying we don't want the href attribute to do anything anymore. We're going to be handling clicks with Javascript now.

This attribute will manifest itself in the rendered HTML page as a CSS class on the `` tag that wraps the button. If you reload the page and then look at the HTML source of the hello button/menu-item, it will look something like:

```
<li id="hello"
  class="selected hello-action"
  >
  <a class=""
    id="hello-link"
    href="#"
    accesskey="h"
    title=""
    data-xf-permission=""
    >
    <span class="action-label">Hello</span>
  </a>
</li>
```

Note

This HTML has been reformatted to fit on this page, but its content should be the same as what is rendered in your application.

3. Create the javascript file. First create a directory "js" inside your application's main directory if it doesn't already exist. Then create a file named `hello.js` inside this directory with the following contents:

```
(function(){
  var $ = jQuery;
  registerXatafaceDecorator(function(node){
    $('li.hello-action > a', node).click(function(){
      alert("You clicked the hello button!!");
      return false;
    });
  });
})();
```

This snippet requires a little bit of explanation, especially if you're new to Javascript and/or jQuery. Some key points to notice here:

- i. The entire contents of the file is wrapped in an anonymous function:

```
(function(){ ...})();
```

This is a common construct in Javascript, and its purpose is to create a private namespace that won't pollute the rest of the application. All variables created inside this function (using the `var` keyword) are local and only accessible inside this function. This is a best practice when creating Javascript files. You never want to add variables to the global namespace unless you intend for the variable to be accessible elsewhere in the application.

Tip

In Javascript, a function is just an object with some special properties that make it callable. Therefore, you can pass functions around just like any other object - copy them into variables, then call the variable as a function. E.g. The following two are equivalent ways to declare a function named `foo`:

```
function foo(){ ... }
```

and

```
var foo = function(){ ... }
```

In both cases, you could subsequently call `foo` as a function by appending `()` to the end:

```
foo();
```

Javascript also supports anonymous functions (i.e. functions with no name). These are useful if you want to pass a function as a callback to another function so that it can be called at a later time. Or, you can simply wrap the anonymous function inside parentheses, and then call the function immediately. E.g. The following are effectively equivalent:

A. Create a function, then call it immediately:

```
var foo = function(){ ...};  
foo();
```

B. Create an anonymous function and call it immediately:

```
(function(){ ... })();
```

- ii. The first line inside the anonymous function assigns the jQuery variable to a shorthand variable `$`.

```
var $ = jQuery;
```

In Xataface, jQuery is included by default so it should always be available to your applications. However the short variable `$` is disabled by default for compatibility with other libraries. With this line, we create a local reference to jQuery in the variable `$` that is not available outside the body of our function (i.e. not available outside the `hello.js` script).

- iii. The `registerXatafaceDecorator()` function is defined in the `plone-javascripts.js` file which is included in all Xataface requests. It allows you to register a callback function that will be called whenever a document fragment is loaded on the page. This is very similar jQuery's `$(document).ready()` callback, except the `ready()` callback is only fired once when the page is finished loading its content. The callbacks that are registered by the

`registerXatafaceDecorator()` function are called whenever new content is added to the page, including if that content is added with AJAX.

- iv. Inside our callback function, we use jQuery to select the hello button, and assign the "click" handler:

```
$('li.hello-action > a', node).click(function(){
```

If you're not familiar with jQuery selectors, you can read more about them at <http://jquery.org> [???]. The gist of this selector is that it is selecting all `<a>` tags that are children of any `` tags with the CSS class "hello-action". The second parameter, `node`, serves as the root node for our query. It is the node that is passed to our callback function by the Xataface decorator as the root node. Had we omitted this parameter, it would have searched the entire document. This would still work but could potentially be unnecessary and create conflicts later if the application evolves to make heavier use of AJAX.

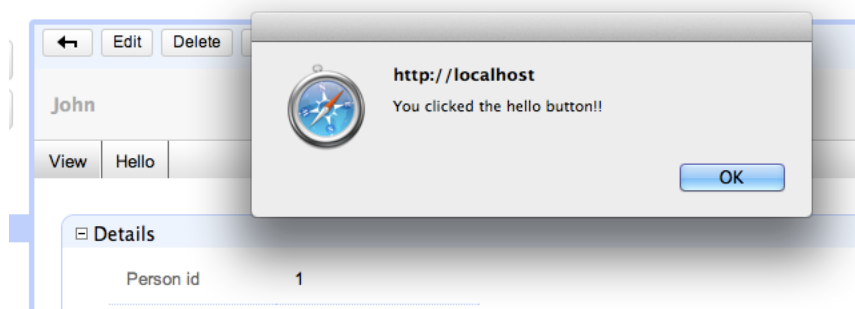
The `.click()` call, assigns an event handler to each of these matches `<a>` tags that is called whenever the link is clicked. The function that is passed as a parameter to `click()` is the function that is to be called on click.

- v. The contents of the `click()` callback function are as follows:

```
alert("You clicked the hello button!!");
return false;
```

The first line simply opens an alert dialog. The second line is important!. You must return false from your click handler to tell the browser not to handle the click event itself. If you don't return false, the user will still be taken to the url of the link on click. In our case we have changed the url to "#", but this will still cause the user to jump to the top of the page if they have scrolled down at all - which is a usability annoyance to be avoided.

4. Try reloading the application, and then click on the "Hello" button. It should pop up with a Javascript alert similar to the following:



This example isn't "useful" per-se, but it should give you enough to get started with Javascript handlers for your actions. Using Javascript to handle events on your menu items gives you much more flexibility for user interaction. For example, you can cause your menus to load dynamic content using AJAX, or prompt the user for additional information before launching your action.

Only Showing the Button When In the People Table

Since our action is designed to work only for the people table, it doesn't make very much sense to have this button appear for records in other tables. The way we have defined this action, however, our button

will appear on the record details view for all tables in the database. In order to prevent this, we'll add the `condition` directive to our action.

The `condition` directive can contain a PHP expression that resolves to a boolean value. If the result is `true` in a given context, then the button will be displayed. If it is `false`, then the button won't be displayed. Xataface will execute this condition just before it renders the action. If you have this action set to be rendered in 9 different locations on the page, then the `condition` expression will be evaluated 9 times.

Our modified "hello" action will look like:

```
[hello]
  category=record_tabs
  url="{ $this->url( '-action=hello' ) }"
  condition="$query[ '-table' ] == 'people' "
```

This example makes use of another context variable, `$query`, which is accessible in all PHP expressions in the `actions.ini` file. It is an associative array of the current query. It is the same as the result of the `Dataface_Application::getQuery()` method - a merging of the `$_POST` and `$_GET` superglobals.

One thing to notice here is that inside the `condition` directive, you don't need to wrap the PHP expression inside braces (`{ }`), whereas inside the `url` directive you do. This is because the default execution context for a `condition` directive is a PHP expression, whereas the default execution context for other types of directives is plain text.

Note

If you are using the `$record->getURL()` method in the `url` directive, then your action would become:

```
[hello]
  category=record_tabs
  url="{ $record->getURL( '-action=hello' ) }"
  url_condition="$record"
  condition="$record and $record->table()->tablename == 'people' "
```

Notice that we don't use the `$query` variable here. Instead we check the table of the `$record` itself. This is because, in this case we are not assuming any particular query context. Instead we are always depending on the current record's table to decide whether the action should appear.

Hiding The Button Based on Permissions

There is nothing more annoying than an application allowing you to click on a button, and then finding out that you don't have permission to perform whatever it is that the button does. It would be better, if you don't have permission to perform the action, for the button to just not appear at all. With the current settings on our `hello` button, users will be able to see it regardless of whether or not they have permission to access the action. We have already set up permissions on the action handler so that if the user tries to access the action, they'll receive a "No Access" message. However, they might still be able to see the button and click on it.

In order to illustrate this point, let's first make some changes to the permission rules in our application. We are currently providing just an all-or-nothing rule where the user is either granted ALL permissions or no permissions depending on whether the current record has first name "John". Let's change this so that the

user has read-only access to all of the records, but only has "hello" permission for records with first name "John". This will make it easier to see how the `permission` directive can affect the display of our button.

In the `tables/people/people.php` file, we'll change the `getPermissions()` method to look like:

```
<?php
class tables_people {
    function getPermissions($record){
        if ( isAdmin() ) return null;
        if ( $record and $record->val("first_name") == "John" ){
            $perms = Dataface_PermissionsTool::READ_ONLY();
            $perms['hello'] = 1;
            return $perms;
        }
        return Dataface_PermissionsTool::READ_ONLY();
    }
}
```

We have made a couple of changes here:

1. In cases where the current record has first name "John", we get the `READ ONLY` permission set, and add the 'hello' permission onto it, and return that. (In our previous version we simply returned `ALL` permissions here).
2. For all other cases we just return `READ ONLY` permissions. We previously had been returning `null` here which deferred the decision to the application delegate class, which, in turn, would return `NO` permissions.

The motivation for these changes is to allow the user to view a record, but not have permission to access our `hello` action.

Now, if we load up our application in the list view of the `people` table, we should be able to see rows for every person in the table. If we click on any of the records, we will be sent to the details view of the record, where we can see our `hello` button along the top. On records where the first name is "John", we can click on the "Hello" button to access our `hello` action. On records where the first name is not "John", we can still click on the "Hello" button, but we'll be greeted by a "Permission Denied" message.

In order to hide the button from records that don't allow the "hello" permission, we just need to add the `permission` directive to our `hello` action definition in the `actions.ini` file:

```
[hello]
category=record_tabs
url="{ $this->url( '-action=hello' ) }"
condition="$query['-table'] == 'people'"
permission=hello
```

Now, if we view a record with first name "John", we'll see our button, and if we visit other records, we won't see it.

Using PHP Expressions In Action Directives

We have already seen a number of examples of PHP expressions inside action directives. The `url` directive contained a reference to the `$this` variable to refer to the `Dataface_Application` object. The `condition` directive referenced the `$query` variable which contains all of the request parameters for

the current HTTP request. There are a limited set of other variables that are available in this execution context as well. These variables are shown in Table 1.1, “Variables available in the execution context of an action directive”.

Table 1.1. Variables available in the execution context of an action directive

Variable Name	Description
<code>\$query</code>	Associative array of query parameters. This is the same as obtaining the query parameters via the <code>Dataface_Application::getQuery()</code> method.
<code>\$this</code>	Reference to the <code>Dataface_Application</code> object.
<code>\$app</code>	Alias of <code>\$this</code> . A reference to the <code>Dataface_Application</code> object.
<code>\$record</code>	<code>Dataface_Record</code> object that has been passed to this execution context. This may be different than the current record of the found set for the current HTTP request. E.g. If the actions are being rendered once per row, where each row contains a different record, then the <code>\$record</code> variable will refer to the record in the row, but <code>\$app->getRecord()</code> will refer to the current record of the request's found set.
<code>\$table</code>	The name of the current table of the HTTP request.
<code>\$tableObj</code>	The <code>Dataface_Table</code> object encapsulating the current table of the HTTP request.
<code>\$relationship</code>	A <code>Dataface_Relationship</code> object encapsulating the current context. This might be the relationship of the current HTTP request or the relationship for a particular action menu as passed from the template.
<code>\$resultSet</code>	The <code>Dataface_QueryTool</code> object encapsulating the current result set of the HTTP request.
<code>\$site_url</code>	The site URL as a string (not including the <code>index.php</code> file). This is the same as the <code>DATAFACE_SITE_URL</code> constant.
<code>\$site_href</code>	The site URL including the file name (i.e. <code>index.php</code>). This is the same as the <code>DATAFACE_SITE_HREF</code> constant.
<code>\$dataface_url</code>	The URL to the Xataface directory. This is the same as the <code>DATAFACE_URL</code> directive.

PHP can be used inside any action directive, but there are a few things you should consider:

1. *Regular vs Condition Directives*: There are two types of action directives: regular directives (e.g. `label`, `url`, etc.), and condition directives (e.g. `condition`, `url_condition`, `label_condition`, etc...).
2. Condition directives have a default execution context of PHP (i.e. they expect to be 100% PHP code) and their expressions should resolve to a boolean value.

3. Regular directives have a default execution context of plain text (i.e. they expect a plain-text string - not PHP). These directives are evaluated by PHP in the same way that a double-quoted string is evaluated in PHP. That is, you can embed variables, and method calls on objects (if they are enclosed in curly braces), but you cannot embed constants, or function calls directly. A good rule of thumb is that all PHP expressions must be enclosed in `{ }` (curly brackets), and also must begin with a dollar sign (`$`).

Tip

If your application won't load after making changes to an action directive, make sure to enable debugging in your `conf.ini` file. Otherwise fatal errors from your actions' PHP expressions won't be reported. Enable debugging by adding the following to the beginning of your `conf.ini` file:

```
debug=1
```

Note that this will also turn on lots of other debugging information so you'll only want to enable this while you are trying to track down fatal errors.

Example "Regular" Directives with PHP

The following are some examples of regular directives that make use of PHP:

1. Setting the `url` of an action to the current site script with some constant GET parameters:

```
url="{ $site_href }?-action=hello&-table=people&person_id=10"
```

This example uses the `$site_href` variable, but the rest of the directive is plain text. Generally it is not recommended to form URLs this way. It is better to either use the `$record->getURL()` method, or the `$app->url()` method to generate a URL based on either the current request context or the current record.

2. Setting the `url` to the current found set, but changing the action to `hello`:

```
url="{ $this->url( '-action=hello' ) }"
```

3. Setting the `url` to the edit form of the current record:

```
url="{ $record->getURL( '-action=edit' ) }"
```

Whenever calling a method on the `$record` object, you should include an `xxx_condition` directive to tell Xataface when it is OK to evaluate the directive. In this case you would add a condition like:

```
url_condition="$record"
```

Which says "Only evaluate the expression in the `url` directive, if `$record` is not null.". This is because it is possible that Xataface will be asked to render the URL for an action, when there are no records found in the current context.

The pattern here is that any directive that could possibly cause a fatal error if one of the variables is null, should be qualified by a corresponding `xxx_condition` directive.

4. Setting the label to "Edit <first name>" where <first name> is the value of the `first_name` field in the current record:

```
label="Edit { $record->val( 'first_name' ) }"  
label_condition="$record"
```


Example "Condition" Directives

The following are some examples of "condition" directives. Note that you can specify a condition directive using the naming convention `xxx_condition`, where "xxx" is the name of a corresponding regular directive. The `xxx_condition` directive would then be used to determine whether or not the xxx regular directive should be processed. The main "condition" directive, is not used to limit execution of any particular directive, it is used to determine whether the action's menu should be rendered in the current context.

1. Only show the menu item if the current table is "people" and the current action is "list" (i.e. the menu item will only show up in the list view of the people table):

```
condition="$query['-table'] == 'people' and $query['-action'] == 'list'"
```

2. The menu item is only shown when in the context of, or attached to, a record of the people table:

```
condition="$record and $record->table()->tablename == 'people'"
```

Note

We had to prefix the main condition with a check to make sure that the `$record` is non-null. If the record is null and you try to perform the call to `$record->table()`, you will get a fatal error when Xataface tries to process the action. In addition, you might not see an error message since Xataface suppresses errors by default when executing action directives. If you do get a crash with no explanation, you can force Xataface to NOT suppress these errors by enabling debug mode in your application. I.e. add the following line to the beginning of your `conf.ini` file:

```
debug=1
```

3. Only show the menu item if the user has permission to perform the `remove related record` action on the context's current relationship in the context's current record.

```
condition="$record and
    $record->checkPermission('remove related record',
        array('relationship'=>$query['-relationship'])
    )"

```

Note

You may be wondering why we wouldn't just use the `permission` directive to achieve this type of check. E.g.:

```
permission="remove related record"
```

The reason is that this would limit access to the menu item to users who are granted the `remove related record` permission on the current `$record`. However, it is possible that the user could be denied access to this permission on the record itself, but granted access to that specific relationship for the current record. Or vice versa. Simply adding the `permission` directive as shown above would be roughly equivalent to the following condition:

```
condition="$record and
    $record->checkPermission('remove related record')"
```

4. Hide the menu item always:

```
condition="false"
```

Troubleshooting Errors in Action Directives

One annoyance to be aware of when developing custom actions with PHP expressions in your directives is that Xataface, by default, suppresses errors when evaluating the PHP expressions. This is to hide all of the notices and warnings that would appear when you refer to query parameters that aren't set, etc.... The problem comes when you actually have a fatal error in your action that halts execution of your PHP script. You could potentially just get a blank white screen (of death) but no error message to let you know what went wrong.

The best way to track down such fatal errors is to enable debugging temporarily. You do this by adding the following to the beginning of your `conf.ini` file:

```
debug=1
```

With this directive enabled, you can reload your page and watch for any fatal errors resulting from the processing of your `actions.ini` files. Notice that this directive will also turn on lots of other debugging information so you'll want to use this option only while tracking down your fatal errors.

"Selected Record" Actions

In the section called "Using Database Data", we created an action that worked on the current record in the found set, and the current found set. The found sets, in those cases, were simply the result of a query on a table. There is another type of found-set that you may want to work with in your actions: the "selected record" set. This is the set of records that a user may have checked off in list view with the intention of performing a specific action on them. Some of the actions that operate on selected records include:

- Copy
- Update
- Delete (At least the delete button in the list view)

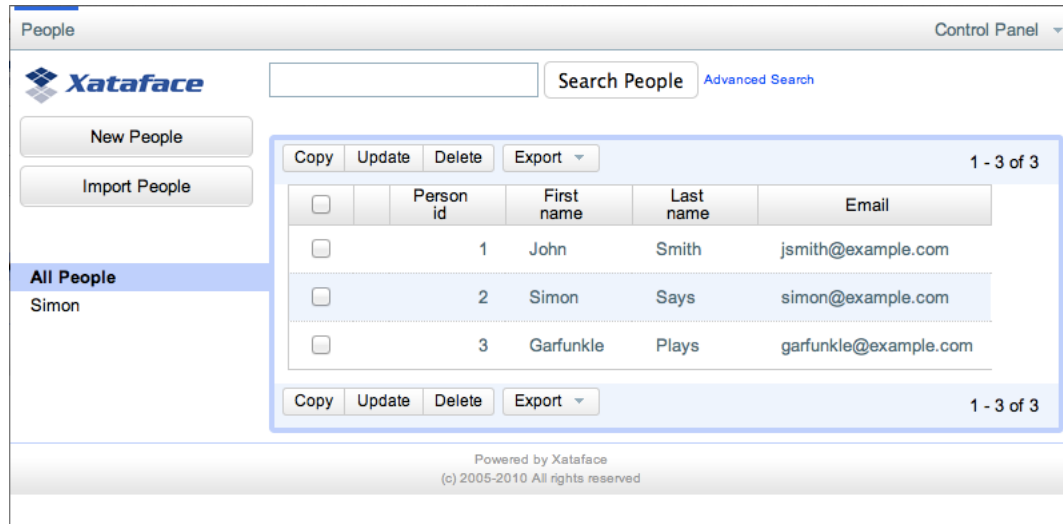
To get a sense of how these actions work, let's load up our application from earlier in this chapter and go to the list view of the people table. Actually, if your permissions are still set to allow only read only access to the people table, you won't see any checkboxes:

The screenshot shows the Xataface application interface for the 'People' table. At the top, there's a 'People' header with a 'Control Panel' dropdown. Below this is the Xataface logo and a search bar with 'Search People' and 'Advanced Search' buttons. On the left, there are buttons for 'New People' and 'Import People', and a sidebar with 'All People' and 'Simon' (selected). The main area displays a table with 3 records. The table has columns: Person id, First name, Last name, and Email. The records are: 1 John Smith (jsmith@example.com), 2 Simon Says (simon@example.com), and 3 Garfunkle Plays (garfunkle@example.com). The 'Simon' record is highlighted. There are 'Export' buttons above and below the table. The footer indicates 'Powered by Xataface (c) 2005-2010 All rights reserved'.

We need to first modify the permissions so that we are granted the "select_rows" permission at the table level (i.e. on no particular record). We can do this by modifying the people delegate class' `getPermissions()` method as follows:

```
function getPermissions($record){
    if ( isAdmin() ) return null;
    if ($record and $record->val('first_name') == 'John'){
        return Dataface_PermissionsTool::ALL();
    }
    $perms = Dataface_PermissionsTool::READ_ONLY();
    $perms['select_rows'] = true;
    return $perms;
}
```

Now, we can refresh the list view of the people table and we'll see a checkbox next to each row of the table:



Now, if you check the box beside one or more rows and click the "Copy" button, you'll be brought to the "Copy Records Form":

Copy Records Form

This form allows you to copy the selected records and update the values of particular fields in the copies.

Selected records:

Title	Person id
Simon	2
Garfunkle	3

Add Field to update:

Please select ... Help

Perform Update Now

! Proceeding with this action will make copies of all selected records. Use caution and care when using this form.

Notice in the left side of the form below the "Selected records" heading, it should list the records that you had selected in the list view. Further, if you were to proceed with this form, the action would effectively modify those selected records, and only those selected records?

This pattern (select some records and perform some action on them) is very common in web-based applications, so you will likely want to replicate this pattern in your own actions at some point. Xataface gives you the tools to do this without having to reinvent the wheel.

Creating a "Selected Records" Action

There are two parts to creating a "selected records" action:

1. The action definition. The action definition in the actions.ini file needs to include one of the applicable CSS classes in its class attribute so that Xataface will know to treat it differently than other actions.
2. Retrieving the selected records inside the action handler. Xataface provides a function `df_get_selected_records()` that will load and return all of the selected records that were passed in the request. Then you can do what you like with these records.

The Action Definition

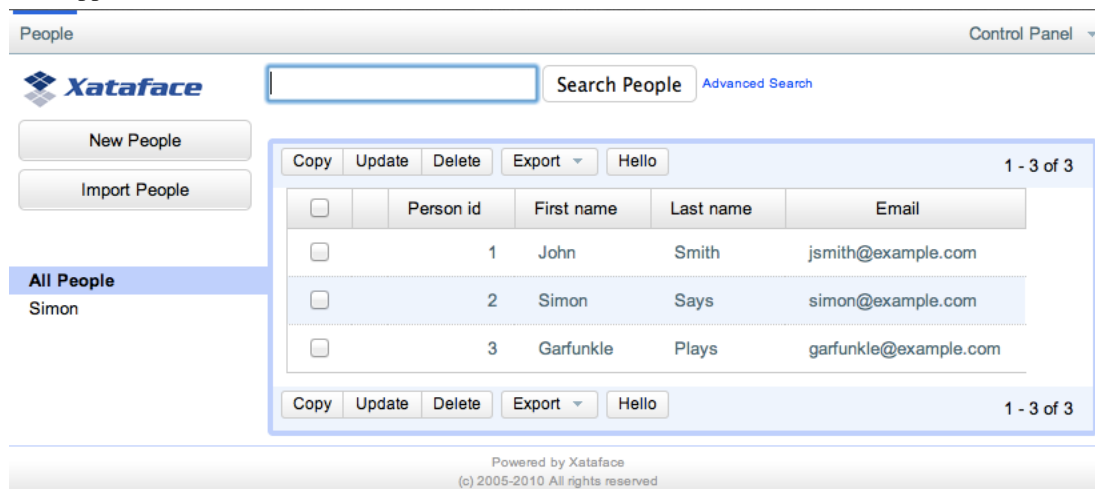
Currently, there are only two categories of actions that will work as selected record actions:

1. `result_list_actions`: These actions show up on the list view.
2. `related_list_actions`: These actions show up on related list views (i.e. the list of related records).

We will set up our "hello" action to operate directly on the people table (not a relationship), so we will set the category to `result_list_actions`. We'll also add the `selected-action` CSS class as part of the class directive so that Xataface knows that this should operate on selected records only. Our new action definition for the hello action is as follows:

```
[hello]
category=result_list_actions
url="{ $this->url( '-action=hello' ) }"
condition="$query['-table'] == 'people'"
permission=hello
class="selected-action"
```

Now, let's reload our application and check out the "list" view to make sure that our action shows up where it is supposed to:



Note

If your `hello` action doesn't appear, it may be due to permissions. We currently have the action set to require the "hello" permission, and the last state of our `getPermissions()` method only allowed the "hello" permission on records with first name "John". If the first record in the result set does not grant the "hello" permission, then the "Hello" button will not show up.

Dealing with "list" permissions like this can be a little tricky because you do want the user to have access to the hello button for the list, but you don't want them be able to access records for which they shouldn't have access.

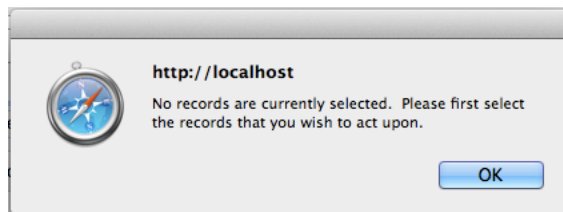
There are a number of solutions to this problem. A couple you might want to consider include:

1. Remove the "permission" directive entirely and rely on explicit permission checking inside the hello action handler (e.g. using `checkPermission()` or `getPermissions()`).
2. Create a separate action for the selected records action button, and a separate permission. You could call the action "hello_selected", and the permission could also be "hello_selected". You could then provide access to `hello_selected` for all records, but still restrict access to the `hello` permission for only records with first name "John". E.g. The action definition would become:

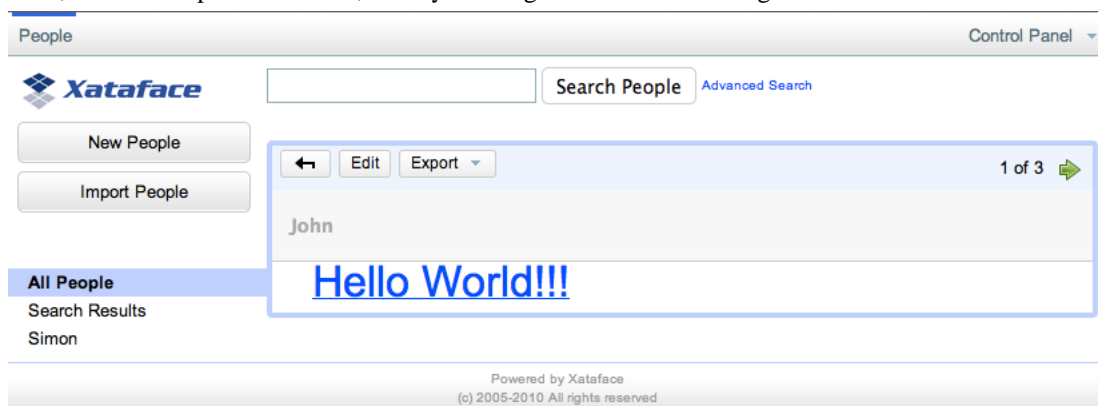
```
[hello_selected]
  category=result_list_actions
  url="{ $this->url( '-action=hello' ) }"
  condition="$query['-table'] == 'people'"
  permission=hello_selected
  class="selected-action"
```

Notice that the `url` directive still points to the "hello" action.

Now, if you click on the "Hello" button without selecting any of the rows in the list, you should receive an alert prompt as follows:



Now, select a couple of the rows, and try clicking the "Hello" button again:



The button works ... sort of. Notice that the action simply loads the hello world action for the first record in the found set. You can try selecting different records and clicking the "Hello" button and it will still just show you the first one in the found set, whether you selected it or not. This is because the `hello` action handler isn't set up yet to load the selected records. The button is sending the information to the action as part of the POST variables, but the action still needs to explicitly do something with that information.

The Action Handler

The `hello` action handler is currently set up to show "Hello World!!!" for the current record of the found set. If we want it to, instead, do something special for all of the records in the found set, we need to make use of the `df_get_selected_records()` function to load the selected records from the query. We're going to change the action slightly to just loop through all of the selected records, and say "Hello <first name>!!!" to each one. The new action handler is as follows:

```
<?php
class actions_hello {
    function handle($params){
        $app = Dataface_Application::getInstance();
        $query = $app->getQuery();
        $records = df_get_selected_records($query);

        df_display(array('people'=>$records), 'hello.html');
    }
}
```

And we'll also change the `hello.html` template as follows:

```
{use_macro file="Dataface_Record_Template.html"}

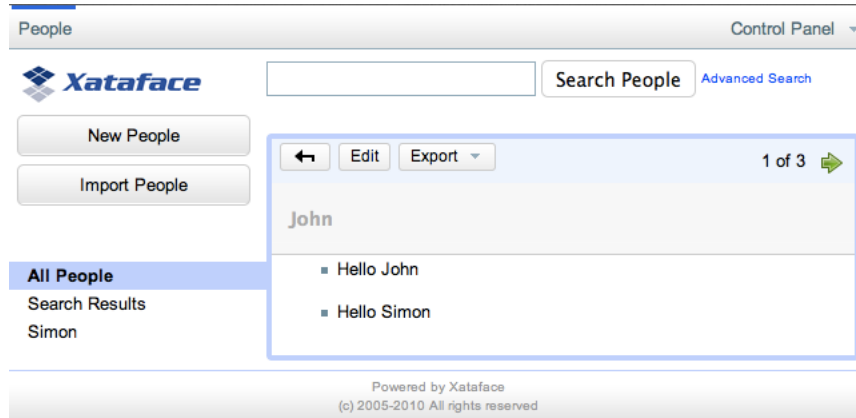
    {fill_slot name="record_content"}
        <ul>
            {foreach from=$people item=person}
                <li>Hello { $person->htmlValue('first_name') }</li>
            {/foreach}
        </ul>
    {/fill_slot}
{/use_macro}
```

Tip

We used the `htmlValue()` method of the `Dataface_Record` class to output the `first_name` field value of each record here. You have probably seen `val()`, `getValue()`, `strval()`, `getValueAsString()`, and `display()` also for retrieving record values. When you are outputting content to a webpage, `htmlValue()` is the preferred method as it does two important things to your value:

1. It escapes HTML special characters to prevent XSS attacks.
2. It applies Xataface permissions by default so that if the user isn't granted the "view" permission on the record, then it will simply display "NO ACCESS".

Now, let's try selecting a couple of records, then clicking "Hello" in our application. You should see something like the following:



You can try selecting different records and clicking "Hello" and you should see the action's output change accordingly.

Building Selected Records Request Manually

The previous section included a little bit of magic: You add a CSS class to your action button, and call `df_get_selected_records()` in your action handler, and voila! You have a "selected records" action. We skipped a couple of steps, though. E.g.:

1. How does the simple act of adding a `selected-action` CSS class to the action menu item make it aware of the selected records?
2. How are the selected records encoded in the HTTP request that is sent to the action handler?

The answer to first question (how the action is transformed) is that there is a little bit of Javascript magic in the Xataface core libraries that looks for actions with the CSS class `selected-action`, and replaces the `onclick` handler with its own. This built-in handler goes through all of the checked rows in the list view and builds a post request with them. It uses the `url` directive of the action (which will be in GET form) and places the parameters inside the POST request so that the request is still sent to the correct action handler.

Selected Records in a POST request

Selected records are included in an HTTP request via any of the following POST parameters:

- `--selected-ids`
- `-selected-ids`

The `--selected-ids` parameter takes precedence over the `-selected-ids` parameter if they are both provided. The value of these parameters is expected to be a new-line-delimited string of record IDs.

Tip

A record ID is a unique identifier for a record that is used by Xataface to refer to a record uniquely. Record IDs follow a logical format:

`tablename?key1=val1&key2=val2&etc...`

For example, if the people table's primary key is the `person_id` field, then the record ID for the people record with `person_id` 10 would be:

```
people?person_id=10
```

The `Dataface_Record` class provides a method called `getId()` which returns the record ID for the record. This is the best way to obtain the record ID for a record. E.g.

```
$person = new Dataface_Record('people',  
    array('person_id'=>10)  
);  
echo $person->getId();
```

Would output:

```
people?person_id=10
```

Xataface also includes some useful functions for dealing with record IDs. `df_get_record_by_id()` can be used to load a record from the database given its ID. `df_parse_uri()` can also be used to parse the record ID into its constituent parts.

If you were to manually make an HTML form that passed selected records to our `hello` action, it would look something like:

```
<form method="POST" action="index.php">  
    <input type="hidden"  
        name="--selected-ids"  
        value="people?person_id=1  
people?person_id=2  
people?person_id=10"  
    />  
    <input type="hidden"  
        name="-action"  
        value="hello"  
    />  
    <input type="hidden"  
        name="-table"  
        value="people"  
    />  
</form>
```

In this case, the `--selected-ids` field includes 3 record IDs. Usually you wouldn't create a manual form like this. However you might find yourself wanting to build an AJAX request with jQuery for a particular action. You could build the equivalent request as follows:

```
var q = {  
    '-action': 'hello',  
    '-table': 'people'  
};  
var ids = [  
    'people?person_id=1',  
    'people?person_id=2',  
    'people?person_id=10'  
];  
q['--selected-ids'] = ids.join('\n');
```



```
jQuery.post(DATAFACE_SITE_HREF, q, function(res){
    // handle response
});
```

When the `hello` action handler receives this request, it uses the `df_get_selected_records()` function to decode the `--selected-ids` POST parameter and load the records that are listed there:

```
$app = Dataface_Application::getInstance();
$query = $app->getQuery();
$people = df_get_selected_records($query);
```

Dynamically Building A "Selected Records" Request

Now that you know how to build a request with selected records, let's put the pieces together and see how we would build a request with the specific records that are currently selected in the result list. Xataface provides some Javascript libraries to help with this. For example, the XataJax module (which comes default with includes some useful functions in its `xatajax.actions.js` file (located in `xataface/modules/XataJax/js/xatajax.actions.js`). One useful function is `XataJax.actions.getSelectedIds()` which will return an array (or newline-delimited string) of record IDs that are current selected inside a specified DOM element.

An easy way, then, to build the `--selected-ids` parameter for a request would be to call `getSelectedIds()`. E.g.:

```
//require <xatajax.actions.js>
//require <jquery.packed.js>
(function(){
    var $ = jQuery;
    var getSelectedIds = XataJax.load('XataJax.actions.getSelectedIds');
    registerXatafaceDecorator(function(node){
        var resultList = $('.resultList');
        var selectedIds = getSelectedIds(resultList);
        var q = {
            '-action': 'hello',
            '-table': 'people',
            '--selected-ids': selectedIds.join('\n')
        };
        $.post(DATAFACE_SITE_HREF, q, function(res){
            // handle the response
        });
    });
})();
```

There are a few things to notice about this snippet:

1. We start out by "requiring" the `xatajax.actions.js` and `jquery.packed.js` scripts. These directives are special directives that are supported by the Javascript tool. They work very similar to the `require` directive in PHP. The Javascript tool will look in the include paths for these scripts and, if found, replace the `require` directives with the contents of the scripts. The XataJax module's `js` directory is in the include path by default, so the `xatajax.actions.js` will be loaded from `xataface/modules/XataJax/js/xatajax.actions.js` (unless you have implemented your own script by the same name in your application's `js` directory).
2. The entire script, excluding the `require` directives is wrapped in an anonymous function that is called immediately. This is to create a private namespace for our script contents.

3. We use the `XataJax.load()` function to retrieve a reference to the `getSelectedIds()` function. This way we don't need to use the full path to the `getSelectedIds()` function every time we call it. See the `XataJax.actions` object API docs [??] for more information about the `getSelectedIds()` function.
4. We wrap all of the script that makes use the DOM inside the `registerXatafaceDecorator()` function so that it will be run when the page is loaded and the DOM is ready. This is similar to the `jQuery(document).ready()` event, except that it has some added benefits when page fragments are loaded through AJAX.
5. We obtain a reference to the `div` tag that wraps the result list table with the call to:

```
$('.resultList')
```

If you look at the HTML source for the list action's generated content to confirm that the wrapper for the table has the `resultList` CSS class assigned to it. We used this as a means of referencing it.

6. We pass the result list wrapper to the `getSelectedIds()` function to obtain an array of string IDs that are currently selected inside the result list.
7. After building the query object, we sent a post AJAX request using jQuery. This particular setup wouldn't work too well for our `hello` action since the action returns a full page (and hence isn't really set up to be used through AJAX). But it gives you a taste of how you could create an AJAX action handler that accepts selected records as part of the query.

Tip

The `XataJax.form` object (defined in `xatajax.form.core.js`), provides a `submitForm()` function that will dynamically create a form and submit it. In cases where you want to create a POST query dynamically but you want to actually post back to the browser (rather than using AJAX), you can use this function instead:

```
//require <xatajax.form.core.js>
(function(){
    var submitForm = XataJax.load('XataJax.form');

    // etc... get the selected ids etc...

    var q = {
        '-action': 'hello',
        '-table': 'people',
        '--selected-ids': selectedIds.join('\n')
    };
    submitForm('post', q);
})();
```

See the `XataJax.form` online API documentation [<http://xataface.com/dox/core/latest/jsdoc/symbols/XataJax.form.html>] for more information.