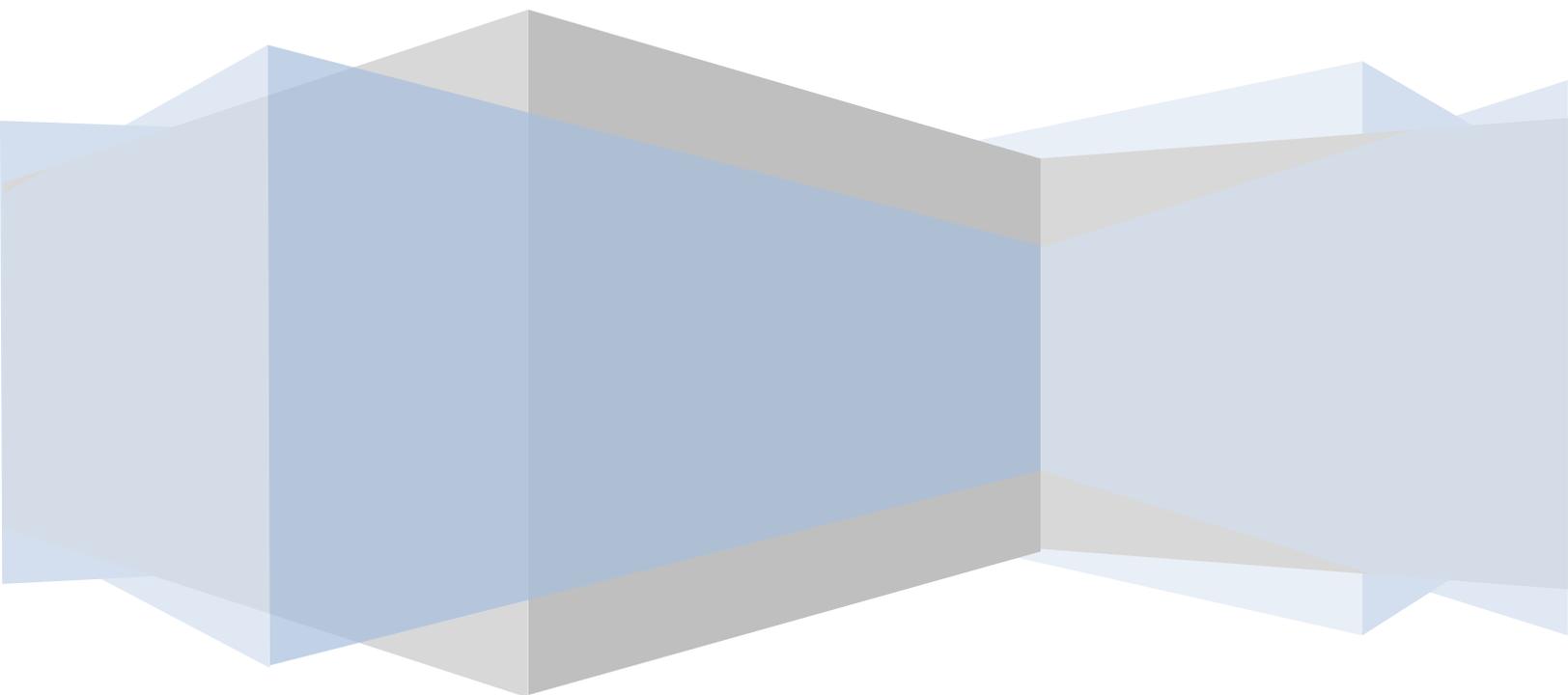


OpenACS.org

# Getting Started With OpenACS

YOUR QUICK GUIDE TO A POWERFUL  
FRAMEWORK.

César Clavería – Rocael Hernández



# GETTING STARTED WITH OPENACS: YOUR QUICK GUIDE TO A POWERFUL FRAMEWORK.

## CONTENTS

Getting Started with OpenACS: your quick guide to a powerful framework.....	2
How this Tutorial is Organized .....	5
Day 1 .....	6
Objectives .....	6
Section 1 .....	6
On this Section You Will Learn.....	6
Installation .....	6
Linux: Installing on a Ubuntu system .....	6
Windows Installer .....	7
Basic Tcl .....	7
What is Tcl?.....	7
Why Tcl? .....	8
Example and Resources .....	8
My First Page .....	8
My First ADP .....	8
Examples.....	8
Passing Variables between pages .....	11
Day 2 .....	13
Objectives: .....	13
Section 2 .....	13
On this Section You Will Learn.....	13
My First Package .....	13

The <i>To Do</i> Application.....	13
Creating a New Package .....	14
Mounting our new package.....	16
Section 3 .....	18
On this Section You Will Learn.....	18
To Do Data model.....	18
Adding some test Entries.....	19
Day 3.....	20
Objectives .....	20
Section 4 .....	20
On this Section You Will Learn.....	20
Creating user accessible pages .....	20
Deleting Items.....	40
Updating the Status .....	41
Section 5 .....	42
On This Section You Will Learn .....	42
Adding a Tcl API to our package .....	42
Day 4.....	45
Objectives: .....	45
Section 6 .....	45
On This Section You Wil Learn .....	45
ACS Objects.....	45
How tu use objects? .....	45
Using Objects .....	46
Changes to the packages Pages.....	50
Section 7 .....	59
On This Section You Wil Learn .....	59
Integrating with other services.....	59

Permissions .....	63
Finishing the transition .....	64
Conclusions .....	66
Appendix A.....	67
OpenACS on Windows .....	67
APPENDIX B.....	68
Connecting to postgresQL on Windows and Ubuntu .....	68

## HOW THIS TUTORIAL IS ORGANIZED:

The tutorial is divided in 4 days (actually 3.5 days! although is known that many people finish it sooner). Each day has two sections, in total there are 7 sections, each section will guide you through a set of activities that will keep you learning OpenACS powerful software development features.

This tutorial was written with an OpenACS system running on an Ubuntu system as reference, so the examples, default directories, screenshots, etc are aimed at an Ubuntu installation, but this do not limit the reach of this tutorial to just Linux at the end of the tutorial you'll find a table with the important directories listed both on Ubuntu and Windows, this can help you if you are using Windows as your base system.

On each section you will find different elements, a typical section will consist of:

- The day number we expect the section to be accomplished.
- What you will learn on this section.
-  Little notes about related topics.
-  Files we will be working on.
- And the "Try It!" icon.



TRY IT!

Sometimes after describing an example your will find a: "Try It!" icon, , this means to go ahead and try what we just did, we will usually give a url to try, this URL will be the one of the reference system, something like: <http://localhost:8000/helloworld> if no url is present then try again the last page we have been working on.

# DAY 1

## OBJECTIVES:

On this first day we want to get you up and running with an OpenACS system and have you to test how to create your very first web pages using OpenACS.

## SECTION 1

### ON THIS SECTION YOU WILL LEARN:

- ✓ [HOW TO SET UP YOUR OWN OPENACS TEST SITE](#)
- ✓ [WHAT IS TCL AND WHERE TO LEARN IT](#)
- ✓ [HOW TO CREATE BASIC WEB PAGES WITH OPENACS](#)

## INSTALLATION

### LINUX: INSTALLING ON A UBUNTU SYSTEM:

You can get the whole system up and running in just a few minutes with it.

Follow these steps to install in Ubuntu, the installer is designed to work with **Ubuntu 8.04 (Hardy Heron)** but it can easily work on newer releases with slight adjustments.

**Follow these steps for Ubuntu 8.04 and Ubuntu 8.10:**

1. Add the following repository to your sources using the package manager or manually add it to the `/etc/apt/sources.list` file:
  - **`deb http://debian.adenu.ia.uned.es/apt hardy main`**
2. Run the command:
  - **`sudo apt-get update`**
3. Install Postgresql:
  - **`sudo apt-get install postgresql-8.2`**
4. Install OpenACS:
  - **`sudo apt-get install openacs`**
5. Start or Restart your OpenACS Service:
  - **`sudo /etc/init.d/openacs start`**

Follow these steps for Ubuntu 9.04:

1. Add this repositories to your sources.list file:
  - **deb http://debian.adenu.ia.uned.es/apt hardy main**
  - **deb http://archive.ubuntu.com/ubuntu/ intrepid universe**
2. Run the command:
  - **sudo apt-get update**
3. Install Postgresql:
  - **sudo apt-get install postgresql-8.2**
4. Install OpenACS:
  - **sudo apt-get install openacs aolserver4-nscache**
5. Start or Restart your OpenACS Service:
  - **sudo /etc/init.d/openacs start**

You can find the most up to date instructions on the installer's Wiki Page:

- ✓ <http://openacs.org/xowiki/ubuntu>

#### WINDOWS INSTALLER:

You can download an all in one installer for Windows (XP and Vista 32 bits), this installer gets you set up with everything you need.

For detailed installation instructions please go to [http://www.friendlybits.com/en/inf\\_tec\\_en/win32openacs\\_en/](http://www.friendlybits.com/en/inf_tec_en/win32openacs_en/)

If you are going to use the windows installer please keep this in mind:



- ✓ It is going to install you 2 services, please use the one named "OpenACS"
- ✓ Before you start the service, start the database server.
- ✓ On Windows Vista you will need to start and stop the database server and the openacs service with administrator privileges.

## BASIC TCL

#### WHAT IS TCL?:

Originally from "Tool Command Language", Tcl is a highly flexible and easy to use programming language, it is the language we will be using to work with OpenACS, so if you do not know it, now it is a great time to try and learn a little bit about it.

---

## WHY TCL?:

The OpenACS toolkit is intended to be used with the AOLServer HTTP server, Tcl is AOLServer's built in scripting language so it is natural to write the applications using Tcl.

---

## EXAMPLE AND RESOURCES:

1. A highly recommended Tcl reference book <http://philip.greenspun.com/tcl/>, it is free and easy to follow, you will learn Tcl in just a few minutes with it, keep it close while developing on OpenACS.

## MY FIRST PAGE

---

### MY FIRST ADP:

ADP stands for AOLServer Dynamic Pages, they are pretty similar to a plain HTML page, but they include a few more tags and extensions to able to handle dynamic information coming from the Tcl page.

The process of creating a user viewable dynamic page on OpenACS is actually divided on at least 2 files a Tcl file and one ADP file, so for a page called "foo" you will have foo.tcl and foo.adp

- ✓ **The adp page is just an HTML page with the necessary markup to display dynamic data.**
- ✓ **The Tcl page handles permissions, logic, calculations and provides the data to the ADP page.**
- ✓ **The naming on these pages is really important, remember that tcl is a case sensitive language.**



There is a third kind of page that could be involved, this third kind defines the database queries to be used, we will not be using them on this tutorial.

---

### EXAMPLES:

On the most basic level you can set a variable on the Tcl file and retrieve it on the ADP, this variable becomes a *datasource* for the ADP page. Open up your favorite editor, we recommend **Notepad++** (<http://notepad-plus.sourceforge.net/uk/site.htm>) on Windows and **Kate** (found under the K Menu on KDE) or **GEdit** (found on the applications menu on Gnome) on a Linux system, and follow these next examples.

---

### HELLO WORLD:



helloworld.tcl:

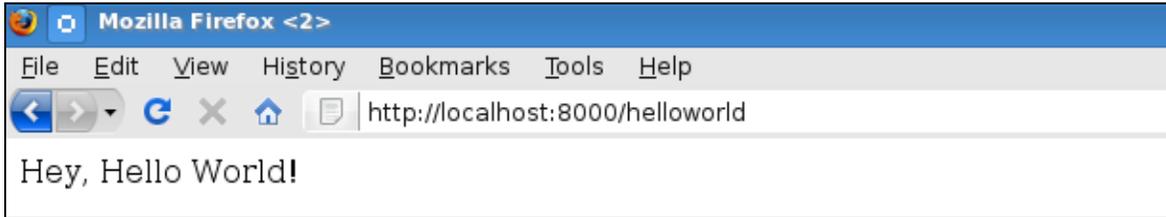
```
set hello "Hello World!"
```



helloworld.adp

```
Hey, @hello@
```

These simple two lines will generate your first page on OpenACS, you can create both files under the `/usr/share/openacs/www` directory on your test site. And when you go to <http://localhost:8000/helloworld> (notice we are not using any file extension) you will see something like this:



View of the page we just created.

**TRY IT!**

<http://localhost:8000/helloworld>

---

### CONNECTION INFORMATION:

On this example we will retrieve some basic connection information using the `ad_conn` procedure and will display it using a simple unordered list.



connection.tcl

```
set user_id [ad_conn user_id]
set url [ad_conn url]
set session_id [ad_conn session_id]
set IP [ad_conn peeraddr]
```



connection.adp

```
<h2>Basic Connection Information:</h2>

<ul>
  <li>User Id: @user_id@</li>
```

```
<li>This URL: @url@ </li>
<li>This session: @session_id@ </li>
<li> IP Address: @IP@ </li>
</ul>
```

Go to <http://localhost:8000/connection> and you will get something like this:

## Basic Connection Information:

- User Id: 568
- This URL: /connection
- This session: 30002
- IP Address: 64.191.80.135

TRY IT!

<http://localhost:8000/connection>

If you notice, right now our last couple of pages do not look quite attractive and there is no title on the pages, they are very simple pages and there is a really easy way to improve their look by adding a couple of lines to the adp page. Let's improve our last example by using the [master](#) tag at the start of the adp page.

Better looking connection.adp:

```
<master>
<property name="doc(title)">Basic Connection Information </property>

<h2>Basic Connection Information:</h2>

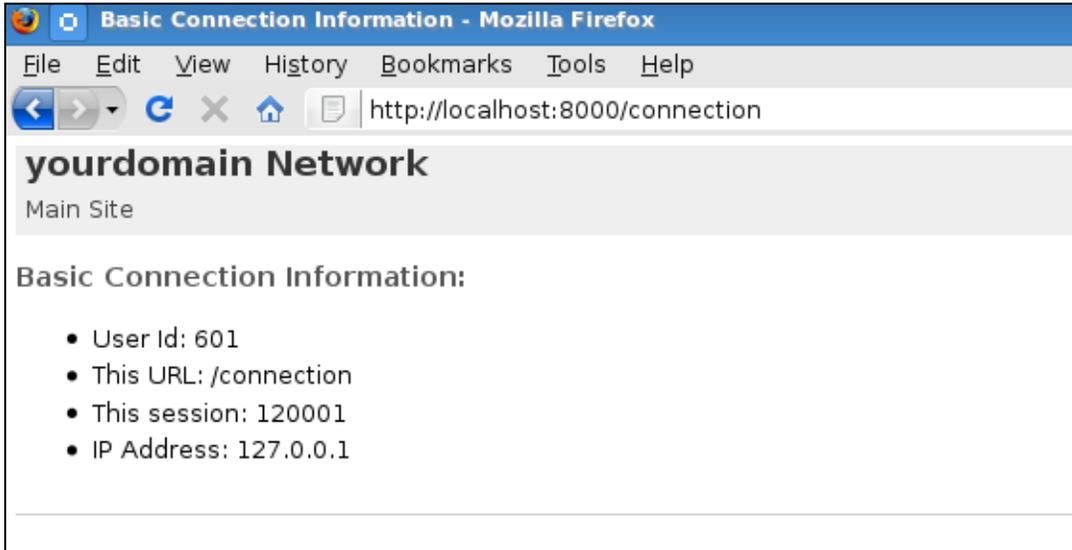
<ul>
<li>User Id: @user_id@</li>
<li>This URL: @url@ </li>
<li>This session: @session_id@ </li>
<li> IP Address: @IP@ </li>
</ul>
```

You should see your page with the site template, the **<master>** tag tells OpenACS to include a header, footer, CSS and few more things to your page, using this you can build sites with a unified look and feel. The **property** tag can set different page attributes, in this case the title of the page.

TRY IT!

<http://localhost:8000/connection>

On your test site the page should look something like this:



The look may change from system to system depending on the site's template, but a basic install should look like the image.

## PASSING VARIABLES BETWEEN PAGES:

To ensure the presence of the required information for the Tcl page OpenACS uses a simple and powerful contract system between the Tcl page and the ADP; this is established by the `ad_page_contract` procedure in the Tcl page. We will build a pretty simple page to illustrate this.

The example will present a simple form for the user to fill out on one page and another to display the information. We will ask for a name and an age and display it on the other page, the name, the age and if the person is an adult.



nameage.tcl

nothing here



nameage.adp

```
<master>

<form action="display-name-age">
  <label for="name">Name</label>
  <input type="text" name="name"/><br />
```

```
<label for="age">Age</label>
<input type="text" name="age"/><br />
<input type="submit">
</form>
```



display-name-age.tcl

```
ad_page_contract {
  This page will display the name and age entered on the nameage page.
  And this page is role is pretty much just to accept and validate the data before displaying it.
} {
  age:integer
  name
}
```



display-name-age.adp

```
<master>
<property name="title">Page for @name@</property>

<ul>
<li>Name: @name@</li>
<li>Age: @age@</li>
</ul>

<if @age@ ge 18>
  @name@ is an adult.
</if>
<else>
  @name@ is still a minor.
</else>
```

**TRY IT!**

<http://localhost:8000/nameage>

## DAY 2

### OBJECTIVES:

Now that you know how to create pages, we want to learn on this day how to create a new package and how to work it inside the system.

## SECTION 2

### ON THIS SECTION YOU WILL LEARN:

- ✓ *HOW TO CREATE YOUR OWN PACKAGE*
- ✓ *HOW TO CONFIGURE A PACKAGE*
- ✓ *WHAT IS THE BASIC STRUCTURE OF A PACKAGE*
- ✓ *HOW TO MOUNT YOUR PACKAGE*
- ✓ *WHAT APPLICATION WE WILL CREATE ON THIS TUTORIAL*

## MY FIRST PACKAGE

Even when using single pages and folders you can accomplish some neat stuff using OpenACS, the real power of the toolkit lies with its packages system. To illustrate this we are going to build a simple package, a "To Do" List application and we will integrate it as a new OpenACS package.

### THE *TO DO LIST* APPLICATION:

The requirements for our *to do list* application are pretty simple, we want to build a basic "To Do" List with the following features:

- ✓ To let the users keep a list of items they need to accomplish before a given date.
- ✓ The users must be able to see and modify only their own items.
- ✓ The information we need to have on each item is:
  - Title
  - The description of the task
  - The due date
  - The item's current status

We need to give the users an interface to:

- ✓ View their list of tasks.
- ✓ Enter a new item to the list.
- ✓ Edit/Delete an item.

---

#### CREATING A NEW PACKAGE:

At a very basic level an OpenACS package is just a directory under the "packages" directory, you can probably find it on **/usr/share/openacs/packages<sup>1</sup>**, with the necessary files (Tcl/adp files) and a file with metadata about the package, but you do not need to write it all yourself, the proper way to create the package is by going in to the "Package Manager" and selecting the option to create a new package.

Go to your site administration: <http://localhost:8000/acs-admin/>

Then follow the link "Developer's Admin" and click on "Package Manager" (<http://localhost:8000/acs-admin/apm>), you should see a page listing all the packages installed on your system:

Scroll all the way down and you will find the link to "Create a new package"

---

<sup>1</sup> On a Windows system: **C:\aolserver\servers\openacs\packages**

profile-provider	Profile Provider	2.3.1	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
ref-timezones	Reference Data - Timezone	5.3.2	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
rss-support	RSS Support	0.3	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
search	Search	5.3.2	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
static-portlet	Static Portlet	2.3.1	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
theme-zen	Zen Theme	2.3.1	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
user-profile	User Profile	2.3.1	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>

- [Create a new package.](#)
- Write new specification files for all installed, locally generated packages
- Load a new package from a URL or local directory.
- Install packages.

**Help**

A package is **enabled** if it is scheduled to run at server startup and is deliverable by the request processor.

After following that link you will be presented with a page to enter the new package information, fill out the form with this information:

<b>Package Key</b>	Todo
<b>Package Name</b>	To Do List
<b>Package Plural</b>	To Do Lists
<b>Package Type</b>	Application
<b>OpenACS Core</b>	Leave it unchecked
<b>Singleton</b>	Leave it unchecked
<b>Auto-Mount URI</b>	Leave it blank
<b>Package URL</b>	Leave the default
<b>Initial Version</b>	0.1d
<b>Version URL</b>	Leave the default
<b>Summary</b>	A little application to keep track of items on a "To do list"
<b>Description</b>	A little application to keep track of items on a "To do list"
<b>Primary Owner</b>	Leave the default
<b>Primary Owner URL</b>	Leave the default
<b>Secondary Owner</b>	not necessary
<b>Secondary Owner URL</b>	not necessary
<b>Vendor</b>	not necessary
<b>Vendor URL</b>	not necessary



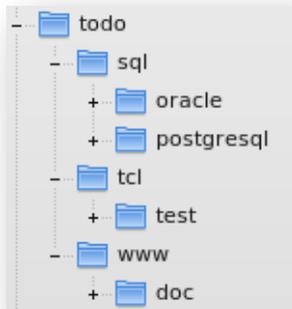
Make sure to leave checked the **"Write a package specification file for this package"** check box.

Now after you click the **"Create Package"** button, the package will be created and installed on your system.

We can check that the package was created by going to the "packages" directory under your OpenACS installation, **/usr/share/openacs/packages**, and looking for a directory named "todo".

It should be: **/usr/share/openacs/packages/todo**

Inside of it you will find the file **todo.info** and the following file structure:



Each file and directory has its own special purpose:

---

#### TODO.INFO:

This is the package specification file, inside of this file is the information that makes this directory a package inside of OpenACS, the package name, parameters, dependencies, etc are defined here, usually you do not need to edit this file directly because it is generated by the Package Manager when there is a change on it.

---

#### SQL:

This directory holds the data model for the package, the initial definition and changes to the data model are defined here. There are two subdirectories here, oracle and postgresql, this is to allow the definition of database specific code, we will be using the postgresql directory only on this tutorial.

---

#### TCL:

The Tcl directory holds the library files containing the definition of procedures related to your package.

---

#### WWW:

These are the public pages of your package, here you will define the pages that the user will interact with, a special service of OpenACS called the request processor maps each request done to an instance of your package to these pages, so you can have your package mounted at different URLs all working with the same code.

You will find this same basic structure on every OpenACS package.

---

### MOUNTING OUR NEW PACKAGE:

Now that our package is created we will mount an instance of it on our "Site Map". Mount means that the pages at its www directory can be accessed by the user.

Head to the administration page, <http://localhost:8000/acs-admin/>, under "Subsite Administration" go to your "Main Site", <http://localhost:8000/admin/> once there look under the "Advanced Features" section for the "Site Map" option.



Usually the URL for this page would be something like: <http://localhost:8000/admin/site-map/>

On this page you are able to see the different "site nodes" that exists on your site, packages, subsites, it is all here. Scroll to the bottom and look for a little form that will let you mount your package, you can even give it any name you want at the moment of mounting, but we will leave it blank this time.

test	Automated Testing	Automated Testing	add folder unmount rename delete parameters permissions
theme-zen	Zen Theme	Zen Theme	add folder unmount rename delete parameters permissions
(+) tutorial	OACS Tutorial	Subsite	add folder unmount rename delete parameters permissions
webdav-support	webDAV Support	webDAV Support	add folder unmount rename delete parameters permissions

Assessment  
 Assessment  
 Bulk Mail  
 Calendar  
 Categories  
 dotLRN  
 dotLRN Homework  
 Evaluation  
 FAQ  
 File Storage  
 Forums  
 News  
 Search  
 Static Portlet  
 Subsite  
To Do List  
 Tutorial  
 User Profile  
 --Install new package--

Click to mount your package.

Select the "To Do List" Package.

»Create new application

»Manage unmounted applications

»Build Site Map

**Services**

- ACS Events
- ACS Mail Lite
- Authentication
- Bootstrap Installer
- Bulk Mail Portlet
- Calendar Portlet
- dotLRN Calendar Applet
- dotLRN Portlet
- FAQ Portlet

After clicking "Mount Package" the new instance of your package will appear on the site map list, you can even browse to it! but, it does not have any content yet.

theme-zen	Zen Theme	Zen Theme	add folder unmount rename delete parameters permissions
todo	To Do List	To Do List	add folder unmount rename delete permissions
(+) tutorial	OACS Tutorial	Subsite	add folder unmount rename delete parameters permissions
webdav-support	webDAV Support	webDAV Support	add folder unmount rename delete parameters permissions

Assessment

You can create a simple "index.adp" or "index.html" file under the www directory of your package to test that it is working.

## SECTION 3

### ON THIS SECTION YOU WILL LEARN:

- ✓ *HOW TO DEFINE A DATA MODEL IN OPENACS*
- ✓ *HOW TO MAKE AN SQL FILE EXECUTE ON PACKAGE INSTALLATION*
- ✓ *WHAT DOES OUR TO DO DATA MODEL CONTAINS*

### TO DO DATA MODEL:

We will need to create at least one new table on the data base to store the information of our "To Do" list.

We need to store:

- ✓ The title of our task.
- ✓ A description.
- ✓ A due date.
- ✓ The state of our to do list (pending, completed, canceled).
- ✓ Who is the owner/creator of the to do list.
- ✓ When we created it.
- ✓ When we modified/updated it.
- ✓ To what package instance it belongs to.

Run the next create table statement on your data base to create the "todo\_item" table:

```
create table todo_item (  
  item_id integer,  
  title varchar(200),  
  description text,  
  status char(1),  
  owner_id integer,  
  due_date date default now(),  
  creation_date date default now(),  
  constraint to_do_list_pk primary key (item_id),  
  constraint to_do_owner_fk foreign key (owner_id) references users  
);
```



Then save it to: `"/usr/share/openacs/packages/todo/sql/postgresql/todo-create.sql"`

By saving it with that name you make sure that if you ever install the "todo" package in any OpenACS system, it will create the todo\_item table during the installation process; right now we will not use this file again, but will come in handy if you need to install the package again on another system.

If you need help running the script, you can check the Appendix B at the end of this tutorial.



You can also specify a file with instructions to be executed when uninstalling the package, you save this to: "/usr/share/openacs/packages/todo/sql/postgresql/**todo-drop.sql**" for now a simple drop table statement will work.

```
drop table todo_item;
```

---

#### ADDING SOME TEST ENTRIES:

Let's test our table by adding a few items directly on the data base.

You will need to know your user id, to find this out go to the Users administration page: <http://localhost:8000/acs-admin/users/> and look for your user details page, there you will find your user's id.

Then run this commands on the data base prompt:

```
insert into todo_item values(acs_object_id_seq.nextval, 'My first item', 'My first item
description', 'p', 568);
insert into todo_item values(acs_object_id_seq.nextval, 'My second item', 'My second item
description', 'p', 568);
insert into todo_item values(acs_object_id_seq.nextval, 'Another item', ' item description', 'p',
568);
```

**TRY IT!**

On psql run both the create table statement and the test values. See appendix B on how to connect.

## DAY 3

### OBJECTIVES:

On your third with OpenACS we want you to be able to finish up the package we started yesterday, have all the user interface set up and be able to use your "To Do" list application.

## SECTION 4

### ON THIS SECTION YOU WILL LEARN:

- ✓ **HOW TO CREATE WEB FORMS WITH OPENACS**
  - **HOW TO USE AD\_FORM FOR:**
    - **ENTERING NEW DATA**
    - **EDITING AN EXISTING ENTRY**
    - **DISPLAYING ONE ENTRY'S INFORMATION**
- ✓ **HOW TO CREATE REPORTS WITH OPENACS**
  - **HOW TO USE THE LIST BUILDER FOR:**
    - **DISPLAYING INFORMATION AS A LIST**
    - **ADD SORTING TO A LIST**
    - **ADD ACTIONS TO A LIST**

### CREATING USER ACCESSIBLE PAGES:

Our next step will be to create the user interface of our package; we need to create the web pages that the user will be interacting with. Please remember to save all of this pages you want the user to be able to access and the **www** directory of your package, this is very important for the package to work properly.

### ADD AND EDIT ITEM PAGE:

We will create our page to add an item, we will do it in a way that we can reuse the same page to edit the item.

You can do a simple HTML form and handle the user inputs yourself, but there is no fun in that. We will do it the OpenACS way, by using the built in procedure `ad_form` that allows you to specify the form fields, the actions to be executed on different states of the form, input validation, etc. You can read all about `ad_form` here: [ad\\_form on the OpenACS API](#)

Let's do the ADP first, you will see the page is really simple:



todo-ae.adp

```
<master>
<property name="doc(title)">@page_title@</property>

<formtemplate id="todo_item_form"></formtemplate>
```



todo-ae.tcl

Let's begin with the `ad_page_contract` for this page and declaring a few variables that we will use later.

```
ad_page_contract {
  This page allows the users to add new items to their to do list or edit existing items.
} {
  item_id:optional
}

set page_title "Add/Edit Todo Item"
set user_id [ad_conn user_id]
```

If you notice we have declared `item_id` as optional parameter on this page, this allows to easily know if we are on edit mode or adding new information, if `item_id` is present then we are editing an existing item, if not, we are adding a new one, `ad_form` is smart enough to know this. We have used `ad_conn` to get information about the current user.

#### FORM WIDGETS:

Let's continue by building our forms widgets, take a look at the next code snippet:

```
ad_form -name todo_item_form -export { user_id } -form {
  item_id:key
  {title:text {label "Task Title"} }
  {description:text(textarea) {label "Description"}}
  {due_date:date(date) {label "Due Date: "} {format {MONTH DD YYYY}} }
  {status:text(select) {label "Status"}
    {options { {"Pending" "p"}
              {"Complete" "c"}
              {"Canceled" "x"}
            }}
}
```

```
}  
}
```

On the first line we are starting to use `ad_form`, the "name" parameter gives a name to our form, this way we can reference it on the ADP or later on the same Tcl file. We also "exported" some variables to the `ad_form`, this takes the variables from our Tcl file and includes them in the form as hidden fields.

```
ad_form -name todo_item_form -export {user_id } -form {
```

After the "-form" switch we can start defining our form's widgets, we start with the "key" widget, this is usually a unique identifier for the item we are adding or editing.

```
item_id:key
```

Next we have a simple text field.

```
{title:text {label "Task Title" } }
```

Our Next line defines a "text area" widget for our task's description.

```
description:text(textarea) {label "Description"}}
```

The next one is a date widget, what this actually does is to insert 2 "selects" and one text field so the user can select the month and day and input a year, this saves you the trouble of handling 3 different widgets for just one piece of information. We specify also the format we wish to use with the date.

```
{due_date:date(date) {label "Due Date: "} {format {MONTH DD YYYY} } }
```

Our last widget is a "select" widget, this one contains an "options" list with all the possible values and labels this select will present to the user, each option is a list of two elements `{<label> <value> }`

```
{status:text(select) {label "Status"}  
  {options { {"Pending" "p"}  
            {"Complete" "c"}  
            {"Canceled" "x" }  
          } }  
}
```

At this point if you go to this page in the browser you will see something like this:

Task Title (required)

Description (required)

Due Date: (required)

Status (required)

If you notice the required label next to each label, this means that `ad_form` is doing some basic validation for you, if any of the required fields has no value, then the form processing will stop and an error message will appear next to each field, like this:

Task Title (required)   
Task Title is required

Description (required)   
Description is required

Due Date: (required)     
Due Date: is required

Status (required)

To make a field optional, you just need to change a little bit the widgets definition, let's do it for the Description field, because sometimes the title says it all.

```
{description:text(textarea),optional {label "Description"}}
```



<http://localhost:8000/todo/todo-ae>

INSERTING NEW DATA:

---

The next section of our form is going to define what we will do with new data, when we are inserting a new item to our to do list. This switch's name is "new\_data" this block contains code that will be executed only when we are inserting a new item, usually we use it with database statements to store the information.

```

-new_data {
db_dml insert_item "
  insert into todo_item
    (item_id, title, description, status, due_date, owner_id)
  values
    (:item_id, :title, :description, :status, to_date(:due_date,'YYYY MM DD HH24 MI SS'),
:user_id)
"
}

```

Note here, for the "due date" value we used the "to\_date" function, the date widget creates a list with the year, month, day, hour, minute and seconds fields, so we can't use it directly.

Right now your ad\_form call should look like this:

```

ad_form -name todo_item_form -export {user_id} -form {
  item_id:key
  {title:text {label "Task Title"} }
  {description:text(textarea),optional {label "Description"}}
  {due_date:date(date) {label "Due Date: "} {format {MONTH DD YYYY}} }
  {status:text(select) {label "Status"}
    {options { {"Pending" "p"}
              {"Complete" "c"}
              {"Canceled" "x"}
            }}
  }
} -new_data {
db_dml insert_item "
  insert into todo_item
    (item_id, title, description, status, due_date, owner_id)
  values
    (:item_id, :title, :description, :status, to_date(:due_date,'YYYY MM DD HH24 MI SS'),
:user_id)
"
}

```

Try it now, and you will be able to insert a new item in the database, you will need to do a query on the database prompt to see the results, for example:

To test your page go to: <http://localhost:8000/todo/todo-ae>

```
openacs=> select * from todo_item;

item_id | title | description | status | owner_id | due_date | creation_date | last_modified_date
-----+-----+-----+-----+-----+-----+-----+-----
2298 | first test | first test description | p | 568 | 2009-06-24 | 2009-04-26 | 2009-04-26
(1 row)
```



At this point you have a page that effectively stores a new item on the data base, you can still do much more, maybe you would like to check the next section to learn how to display your information, or continue reading this section to learn how to edit the an existing item.

### EDITING AN EXISTING ITEM:

---

To edit an existing item we need to do 2 things, first, get all the information about the existing item and fill the values of the form's widgets, and then update the database when the form is submitted.

#### 1. GETTING THE EXISTING DATA

We get the existing information by using the "select\_query" switch, on the body of this block we do a query that returns a single row with all the information of user editable fields, this values will be transferred to the form widgets. This switch can only be used if the form contains a "key" element. This is the select query for our form:

```
-select_query {
  select title,
         description,
         to_char(due_date, 'YYYY MM DD') as due_date,
         status
  from todo_item
  where item_id = :item_id
         and owner_id = :user_id
}
```

Get the item\_id from the database (like we did a couple of sections back) and append it to the form page url, now you should be able to see the fields filled with data when you enter:

Task Title (required)

Description

Due Date: (required)

Status (required)

## 2. UPDATING THE ITEM

Now that we have the data on the form we can edit it, but we also need to be able to store the edited information, for this we use another switch, the “edit\_data” switch, it works pretty much in the same way as new\_data does, the only difference is that it runs only when we are updating an item.

```
-edit_data {
db_dml update_item "
  update todo_item
    set title = :title,
        description = :description,
        status = :status,
        due_date = to_date(:due_date,'YYYY MM DD'),
  where item_id = :item_id and owner_id = :user_id "
}
```



Remember to add the where clause to make sure you are updating only the item you want to update, in this case we added a couple of other checks to make sure we are updating only items that belong to us.

### AFTER SUBMIT:

Sometimes, after processing correctly the form you may wish to do some more calculations or simply redirect the user to another page, we can easily do this in the “after\_submit” block. Here we will use it to direct the user to this package's index page.

```
-after_submit {
  ad_returnredirect index
  ad_script_abort
}
```

## FORM MODES:

---

At last in this section we will add one more trick to our form, a display mode, this way you can present the item's information in a non-editable way to the user.

To do this you just need to add one more switch to the `ad_form` call, like this:

```
ad_form -name todo_item_form -export { user_id } -mode $form_mode -form {
```

We added the `form_mode` variable as part of the `ad_page_contract`, with a default value of "edit", it can also be "display" to display the information in a non-editable way.



`todo-ae.tcl`

The complete Tcl file for `todo-ae.tcl` looks like this:

```
ad_page_contract {
  This page allows the users to add new items to their to do list or edit existing items.
}{
  item_id:optional
  {due_date ""}
  {form_mode "edit" }
}

set page_title "Add/Edit Todo Item"
set user_id [ad_conn user_id]

ad_form -name todo_item_form -export {user_id } -mode $form_mode -form {
  item_id:key
  {title:text {label "Task Title" } }
  {description:text(textarea),optional {label "Description"}}
  {due_date:date(date) {label "Due Date: " } {format {MONTH DD YYYY} }}
  {status:text(select) {label "Status"}
    {options { {"Pending" "p"}
              {"Complete" "c"}
              {"Canceled" "x" }
            }}
  }
} -select_query {
  select title,
```

```

        description,
        to_char(due_date, 'YYYY MM DD') as due_date,
        status
    from  todo_item
    where item_id = :item_id
        and owner_id = :user_id
}-new_data {
db_dml insert_item "
    insert into todo_item
        (item_id, title, description, status, due_date, owner_id, package_id)
    values
        (:item_id, :title, :description, :status, to_date(:due_date,'YYYY MM DD'), :user_id)"
}-edit_data {
db_dml update_item "
    update todo_item
        set title = :title,
            description = :description,
            status = :status,
            due_date = to_date(:due_date,'YYYY MM DD'),
        where item_id = :item_id and owner_id = :user_id "
}-after_submit {
ad_returnredirect index
ad_script_abort
}

```

---

## INDEX PAGE:

We will create the index page as a list of all the “To do” items we have created.

To create the list we will use the **List Builder**, a powerful tool from the OpenACS templating system, it allows us to easily create lists with a great deal of functionalities. The main procedure of the list builder is [template::list::create](#) we can do many things with this procedure, but right now we will limit ourselves to the basics.

We want this page to show a table with this information:

Title	Due Date	Status	Actions
The title for the item goes here.	April 28 2009	Pending	View Edit Delete Mark Completed
another item	April 1 2010	Pending	View Edit Delete Mark Completed
Finish the tutorial	April 27 2009	Pending	View Edit Delete Mark Completed

The way to use the list builders is similar to the way we used **ad\_form** in the previous section, it is just one call to a procedure and different switches affect the outcome and behavior of the list, also, associated to the list there is a **multirow**, we will create this by using the [db\\_multirow](#) function from the database API.

The ADP page is very simple, so Let's get that out of the way:



index.adp

```
<master>
<property name="doc(title)">@title@</property>

<listtemplate name="todo_list"></listtemplate>
```

### The ad\_page\_contract:

---

Right now we do not have anything on our ad\_page\_contract, but as we build functionality into the list we will add one or two more things.

```
ad_page_contract {
  This page will display a list of to do items belonging to the current user.
} {
}
```

### Defining variables:

---

We will define a couple of variables we will use later, this are just some context variables for the query and the adp page.

```
set title "My To Do List"
set user_id [ad_conn user_id]
```

### Defining the list:

---

We will start by defining the list and the first couple of switches:

```
template::list::create -name todo_list \
-multirow todo_list_mr \
```

The first line calls the template::list::create procedure, names our list, the name is necessary to be able to call it on the ADP page, we also define here the name of the multirow that will be the source of the list elements.

## Defining the elements:

---

We define the elements of the list inside the "elements" switch, each element follows the same basic format:

```
element_name {  
  element_option option_value  
  element_option option_value  
}
```

There are many options and you do not always need to use them, you can read all about them here: [template::list::element::create on the OpenACS API](#).

Let's start adding our elements:

A simple:

```
title {  
  
}
```

would work, because the list builder knows you want to output the value of the title field on each row returned on the multirow so you do not need any additional options, but we can do it better than that.

To add a column header use the label option:

```
label "Task"
```

To make the content of the element, in this case the task's title, link to another page use the link\_url\_col option

```
link_url_col <name of a variable with the url>
```

To add additional options to the link use the link\_html option

```
link_html {title "You can create a title text for the link here" }
```

Our complete element for the task's title looks like this:

```
title {  
  label "Task"  
  link_url_col item_url
```

```
link_html {title "Click to view this item details" }  
}
```

The rest of the elements can be defined in a similar way, this would be the complete list builder call for this list:

```
template::list::create -name todo_list \  
-multirow todo_list_mr \  
-elements {  
  title {  
    label "Task"  
    link_url_col item_url  
    link_html {title "Click to view this item details" }  
  }  
  due_date_pretty {  
    label "Due Date"  
  }  
  status_text {  
    label "Status"  
  }  
  creation_date_pretty {  
    label "Creation Date"  
  }  
  view {  
    display_template "View"  
    link_url_col item_url  
  }  
  delete {  
    display_template "Delete"  
    link_url_col delete_url  
  }  
  completed {  
    display_template "Mark Completed"  
    link_url_col completed_url  
  }  
  cancel {  
    display_template "Cancel"  
    link_url_col cancel_url  
  }  
}
```

But we are not done yet with the list, we need to define its data source, the multirow, for this we will use the `db_multirow` procedure.

### The list's datasource:

---

The `db_multirow` proc creates a multirow object, this multirow object will contain the results of the database query we will pass it.

The usual way to use `db_multirow` is like this:

```
db_multirow -extend { <list of extra variables to be part of the multirow> } <multirow_name>
<multirow statement name> {
  sql query to retrieve the data
} {
  block of Tcl code that will be executed for each row of the results, it is useful for setting the
  variables from the extend switch
```

Our `db_multirow` is defined like this:

```
db_multirow -extend { item_url delete_url cancel_url completed_url status_text } todo_list_mr
todo_list_mr \
"select item_id,
      title,
      due_date,
      to_char(due_date, 'Month DD YYYY ') as due_date_pretty,
      creation_date,
      to_char(creation_date, 'Month DD YYYY ') as creation_date_pretty,
      status
from todo_item
where owner_id = :user_id"
```

We have extended our multirow with the `item_url`, `delete_url`, `cancel_url`, `completed_url` and `status_text` variables to make the list more useful, we will need to define this in the optional Tcl code block.

The item url will be pretty simple, remember the little "mode" fix we did to the add and edit page just before starting with the list? well, we did it thinking of this, now all we need to do is to tell the add and edit page to show us the information on "display" mode.

```
set form_mode display
set item_url "todo-ae?[export_vars -url { item_id form_mode }]"
```

The [export\\_vars](#) procedure takes care of creating the query string for the URL, it only needs a list of variables it will include on the URL.

The status we store on the database is just one character, so it is not very helpful to display that, we will create a new variable to hold a better description for us. The tcl's switch statement comes handy for that.

```
switch $status {
  p {set status_text "Pending"}
  c {set status_text "Completed"}
  x {set status_text "Canceled"}
  default {set status_text "Unknown" }
}
```

The delete, completed and cancel links are done pretty much the same way, they link to another page that does the actual update and then we return to the list. We will create these pages later, but right now we can just create the links to them.

```

set return_url [util_get_current_url]
set delete_url "todo-delete?[export_vars -url {item_id return_url}]"

if { $status != "c" } {
    set new_status completed
    set completed_url "todo-update-item?[export_vars -url {item_id new_status return_url}]"
}
if { $status != "x" } {
    set new_status canceled
    set cancel_url "todo-update-item?[export_vars -url {item_id new_status return_url}]"
}

```

If everything has worked so far, you should see a list that looks like this:

Task	Due Date	Status	Creation Date				
<a href="#">First Todo Item</a>	April 17 2009	Pending	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>	<a href="#">Cancel</a>
<a href="#">install test system</a>	April 26 2009	Completed	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>	<a href="#">Cancel</a>
<a href="#">Dinner</a>	April 26 2009	Pending	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>	<a href="#">Cancel</a>
<a href="#">finish up tutorial</a>	April 27 2009	Pending	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>	<a href="#">Cancel</a>

**TRY IT!**

at: <http://localhost:8000/todo/>

At this point you have a pretty functional list with your "To Do" items on it, but we can improve it a little bit by adding the ability to sort each column.

## Adding Sorting:

---

To add the ability to sort the different columns we will need to make just a few changes, this are:

1. Adding a new variable on the ad\_page\_contract.

We add a new optional variable, this is the variable that will hold on what column we are doing the sorting, so now our ad\_page\_contract looks like this:

```
ad_page_contract {
  This page will display a list of to do items belonging to the current user.
} {
  orderby:optional
}
```

2. Adding the orderby block to the list builder.

Each element on the orderby block follows the same format: <element name> { <order by clause> } the list builder takes care of knowing when to do ascending or descending order.

```
-orderby {
  title {orderby title}
  due_date_pretty {orderby due_date}
  status_text {orderby status}
  creation_date_pretty {orderby creation_date}
}
```

You can read more about the orderby block here: [orderby on the OpenACS API](#).

3. Adding the order by clause to our db\_multirow's query.

There are two steps involved here:

1. **Getting the order by clause.**

We do this by checking that the orderby variable of the ad\_page\_contract it is not empty, if it is not empty then it means we are sorting a column, we will get the clause into a variable that we will later pass to our query, in the case the orderby variable is empty we will set a default order.

```
if {[exists_and_not_null orderby]} {
  set orderby_clause "ORDER BY [template::list::orderby_clause -name todo_list]"
}
```

```

} else {
  set orderby_clause "ORDER BY due_date asc"
}

```

## 2. Adding the orderby to our query.

We will simply add the `orderby_clause` variable to our query, so our `db_multirow` now looks like this:

```

db_multirow -extend { item_url delete_url cancel_url completed_url status_text } todo_list_mr
todo_list_mr \
"select item_id,
      title,
      due_date,
      to_char(due_date, 'Month DD YYYY ') as due_date_pretty,
      creation_date,
      to_char(creation_date, 'Month DD YYYY ') as creation_date_pretty,
      status
from todo_item
where owner_id = :user_id
$orderby_clause
" {
  ...
}

```

After all of this is done you will notice that each column header is now a link, by clicking on those links you will sort the list by that column, clicking again the same link will resort the list on a different order.

<a href="#">Task</a> ⚡	<a href="#">Due Date</a> ⚡	<a href="#">Status</a> ⚡	<a href="#">Creation Date</a> ⚡
<a href="#">First Todo Item</a>	April 17 2009	Pending	April 26 2009

### Adding an Action Button:

The action buttons appear on top of the list, they are usually used to provide the user an easy way to reach a function that will affect the list, for this example we will add an action button that leads us to add a new item to our list.

To do this we only need to add another block to our list builder statement, we need the "actions" block. The format for each action is pretty simple, but with many actions it may get hard to read. The format is:

```
-actions {
  <Action label> <Action URL> <Action title text>
  <Action label> <Action URL> <Action title text>
}
```

Our action block looks like this:

```
-actions {
  "Add New Task" "todo-ae" "Click here to add a new item to the list"
}
```

The action button will appear at the top of the list and takes you to the "todo-ae" page, when position the cursor on the button the text "click here to add a new item to the list" will appear momentarily.

Add New Task

<a href="#">Task</a>	<a href="#">Due Date</a>	<a href="#">Status</a>	<a href="#">Creation Date</a>	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>
<a href="#">First Todo Item</a>	April 17 2009	Pending	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>
<a href="#">install test system</a>	April 26 2009	Completed	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>
<a href="#">Dinner</a>	April 26 2009	Pending	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>
<a href="#">finish up tutorial</a>	April 27 2009	Pending	April 26 2009	<a href="#">View</a>	<a href="#">Delete</a>	<a href="#">Mark Completed</a>

At this point you have a basic, but functional, to do list application, built using a couple of the tools OpenACS provides. Next is the final index.tcl file with all of the functionality we built in the last few sections.



index.tcl

This is the complete **index.tcl** file for our package.

```
ad_page_contract {
  This page will display a list of to do items belonging to the current user.
} {
  orderby:optional
}
```

```
set page_title "My To Do List"
set user_id [ad_conn user_id]

template::list::create -name todo_list \
-multirow todo_list_mr \
-elements {
  title {
    label "Task"
    link_url_col item_url
    link_html {title "Click to view this item details" }
  }
  due_date_pretty {
    label "Due Date"
  }
  status_text {
    label "Status"
  }
  creation_date_pretty {
    label "Creation Date"
  }
  view {
    display_template "View"
    link_url_col item_url
  }
  delete {
    display_template "Delete"
    link_url_col delete_url
  }
  completed {
    display_template "Mark Completed"
    link_url_col completed_url
  }
  cancel {
    display_template "Cancel"
    link_url_col cancel_url
  }
} -orderby {
title {orderby title}
due_date_pretty {orderby due_date}
```

```

status_text {orderby status}
creation_date_pretty {orderby creation_date}
}-actions {
  "Add New Task" "todo-ae" "Click here to add a new item to the list"
}

if {[exists_and_not_null orderby]} {
  set orderby_clause "ORDER BY [template::list::orderby_clause -name todo_list]"
} else {
  set orderby_clause "ORDER BY due_date asc"
}

db_multirow -extend { item_url delete_url cancel_url completed_url status_text } todo_list_mr
todo_list_mr \
"select item_id,
      title,
      due_date,
      to_char(due_date, 'Month DD YYYY ') as due_date_pretty,
      creation_date,
      to_char(creation_date, 'Month DD YYYY ') as creation_date_pretty,
      status
from todo_item
where owner_id = :user_id
$orderby_clause

" {

set form_mode display
set item_url "todo-ae?[export_vars -url { item_id form_mode }]"

switch $status {
  p {set status_text "Pending"}
  c {set status_text "Completed"}
  x {set status_text "Canceled"}
  default {set status_text "Unknown" }
}

set return_url [util_get_current_url]
set delete_url "todo-delete?[export_vars -url {item_id return_url}]"

```

```

if { $status != "c" } {
    set new_status completed
    set completed_url "todo-update-item?[export_vars -url {item_id new_status return_url}]"
}
if { $status != "x" } {
    set new_status canceled
    set cancel_url "todo-update-item?[export_vars -url {item_id new_status return_url}]"
}
}

```

**TRY IT!**

<http://localhost:8000/todo/>

#### DELETING ITEMS:

We will create a very simple page for deleting the items, it will take an item id, delete it and return the user to the previous page. We only need the Tcl file for this page.



todo-delete.tcl

```

ad_page_contract {
    This page will delete an item from the todo list package and then return the user
    to the specified return URL.
} {
    item_id
    return_url
}

set user_id [ad_conn user_id]
db_dml delete_item "delete from todo_item where item_id = :item_id and owner_id = :user_id"
ad_returnredirect $return_url

```

---

## UPDATING THE STATUS:

The page for updating the items' status works in pretty much the same way, just the dml operation is different this time.

```
ad_page_contract {
  This page will update an item's status from the todo list package and then return the user
  to the specified return URL.
}{
  item_id
  new_status
  return_url
}

set user_id [ad_conn user_id]
switch $new_status {
  "pending" {set status_code p}
  "completed" {set status_code c}
  "canceled" {set status_code x }
  default {set status_code p }
}

db_dml update_status "update todo_item
    set status = :status_code
    where item_id = :item_id
    and owner_id = :user_id"

ad_returnredirect $return_url
```

## SECTION 5

ON THIS SECTION YOU WILL LEARN:

- ✓ HOW TO DEFINE YOUR OWN PROCEDURES ON OPENACS
- ✓ HOW TO CREATE A FILE THAT CONTAINS THE PROCEDURES
- ✓ HOW TO MAKE OPENACS RELOAD YOUR FILIN MEMORY YOUR FILES / PROCEDURES

ADDING A TCL API TO OUR PACKAGE:

OpenACS is not limited to user visible pages, in addition each package can define their own procedures. Remember the /tcl directory on our package? We will use that now.

Create a new file called **todo-procs.tcl** in **/usr/share/openacs/packages/todo/tcl/** and paste the following code there.



todo.tcl

```
ad_library {
    Procs for the To Do list package.
}

namespace eval todo {}

ad_proc -public todo::get_status_label { status } {
    switch $status {
        p {
            set status_text "Pending"
        }
        c {
            set status_text "Completed"
        }
        x {
            set status_text "Canceled"
        }
        default {
            set status_text "Unknown"
        }
    }
}
```

```

return $status_text
}

ad_proc -public todo::get_status_code { status_text } {
    set status_text [string tolower $status_text]
    switch $status_text {
        "pending" {
            set status_code p
        }
        "completed" {
            set status_code c
        }
        "canceled" {
            set status_code x
        }
        default {
            set status_code p
        }
    }
    return $status_code
}
}

```

The first few lines define this tcl file as a new library and provide documentation for its purpose by using the [ad library procedure](#). Next we define a new **namespace** for our procedures, this way we make sure any procedure we define here will not have conflicts with another package's procedure with the same name, but different namespace.

Next we defined two different procedures to encapsulate some code we did earlier, we have set their scope as public so any package can use them.

But we cannot still use the package, we will need to tell package manager that there is a new procedure file and that it needs to load it. Go to the package manager <http://localhost:8000/acs-admin/apm> and look for our package on the list, and click on "reload changed"

<a href="#">search</a>	<a href="#">Search</a>	5.4.3	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>
<a href="#">todo</a>	<a href="#">To Do List</a>	0.1d	Enabled	Locally	<a href="#">view files</a>   <a href="#">watch all files</a>   <a href="#">reload changed</a>

Now the procedures have been loaded into the system and we can use them, it is a good idea to mark the file to be *watched* if we are going to be working on it.

Marked the following file for reloading:

- packages/todo/tcl/todo-procs.tcl ([watch this file](#))

If you know you're going to be modifying one of the above files frequently, select the "watch this file" changed.

Now you can replace the code on the index.tcl page and todo-update-status.tcl for a call to the corresponding procedures, like this. Find the correct place and replace it.



on index.tcl:

```
set status_text [todo::get_status_label $status ]
```



on todo-status-update.tcl:

```
set status_code [todo::get_status_code $new_status]
```

**TRY IT!**

## DAY 4

### OBJECTIVES:

On your fourth day working with OpenACS we want you to learn about the ACS Objects system and how your packages and items can be a part of it so your applications can take full advantage of the powerful services and modules that OpenACS presents. We will learn integrating OpenACS Objects into your "To Do" list application.

## SECTION 6

### ON THIS SECTION YOU WILL LEARN:

- ✓ **WHAT ARE ACS OBJECTS**
- ✓ **HOW TO CREATE AND USE ACS OBJECTS**
- ✓ **WHAT BENEFITS WE HAVE USING ACS OBJECTS**
- ✓ **WE WILL TRANSFORM OUR "TO DO" ITEMS INTO AN OBJECT**
- ✓ **WE WILL ADD OBJECT FUNCTIONALITY TO OUR PACKAGE**

## ACS OBJECTS

OpenACS gives us the option to use its objects system to integrate our application to the system and be able to take advantage of many of the different services that OpenACS provides.

The motivation behind the existence and use of ACS objects is the fact that while developing OpenACS applications and modules you will find that many of them share some common characteristics, and needs, such as:

- User information related to a given database entry
- Roles and permissions over those database entries
- Some might need a versioning system for the entries
- Notifications
- Relate comments, ratings and other possible services
- Multiple instances of the application to be used by different groups within the same website

### HOW TO USE OBJECTS? :

Using the ACS Objects it is simple but it requires you to take some extra steps while defining your data model, this is what you need to do.

1. Create a new object type
  - Object types are similar to classes on an object oriented programming language

2. Define a table for your application
  - This table is for all the application specific information about your object
  - This table's primary key column must reference the **object\_id** in the **acs\_objects** table that is the central table that manages the ACS objects
3. Define application specific procedures
  - This can be either tcl procedures or stored procedures on the database
4. Make your application "object aware"
  - With just a few changes we can make our "To Do" application to be aware of the existence of our objects

Let's start now to modify our "todo" application to use ACS Objects and integrate it better with OpenACS.

---

## USING OBJECTS:

---

### PREPARING OUR PACKAGE:

We will work upon the same package we built in the last few sections, some new functionality will be added and the ability to create and manage different package instances will be explored.

One of the first things we'll do is to drop our current table, execute this command on psql:

```
drop table todo_item;
```

Do not worry, we will rebuild it again and better than ever.

---

### DEFINING THE DATA MODEL:

Remember that we store our data model files on **/usr/share/openacs/packages/todo/sql/postgresql/**



[todo-create.sql](#)

This file is the one that will be loaded by the system when we install our package on another OpenACS installation, in this case since the package is already installed, we will define and then load it manually to the database.

This file serves two purposes:

- ✓ To create our applications data model table.
- ✓ To register a new **object type** on the system.

The first section of the files defines our new table, if you notice we have eliminated many of the previous fields we were using, that is because the objects system will keep track of the package id, creation user, creation date, etc. Also notice that our item\_id now points to the **acs\_objects** table.

The second part is a little bit more interesting, here we will create and execute a function that will register our new object type and will define it as using our table "todo\_item" as the table that holds its information.

```

create table todo_item (
  item_id integer,
  title varchar(500),
  description text,
  status char(1),
  due_date date default now(),
  constraint todo_item_pk primary key (item_id),
  constraint todo_item_fk foreign key (item_id) references acs_objects(object_id)
);

-- Here is the function that will register a new object type for our package, in order the
-- parameters are:
-- Name of the new object_type
-- "Pretty" name of the object type, this is an human readable name for our new type
-- Pretty plural, our pretty name in plural form
-- Supertype, this object parent type, usually it is acs_object for a new kind of object
-- Table name, the name that holds the data model for this object
-- ID column, the column of our data model table primary key
-- Package Name, our package's name
-- The last 3 parameters are not relevant at this time

create function inline_0 ()
returns integer as '
begin
  PERFORM acs_object_type__create_type (
    "todo_item",
    "To Do Item",
    "To Do Items",
    "acs_object",
    "todo_item",
    "item_id",
    "todo",
    "f",
    null,
    null
  );

  return 0;
end;' language 'plpgsql';

```

```
select inline_0 ();

drop function inline_0 ();

\i packages/todo/sql/postgresql/todo-package.sql
```

The last line simply calls a new sql file where we will define a couple of functions to deal with our new object and the `acs_objects` table:



`todo-package.sql`

We will define two Postgresql functions to handle our objects, one to create a new object and one to delete it. Those functions **must** exist in order to comply with the OpenACS standards and expect ACS Object system behavior.

- ✓ **todo\_obj\_item\_\_new:** This function takes as parameters all the fields that we need to handle our to do list items, it uses some of them to create a new object and just leaves the application specific ones to be added to the `todo_item` table we defined earlier. Noticed that we must tell the function that creates the new object what type of object are we creating.
- ✓ **todo\_obj\_item\_\_delete:** This function takes just one argument, the id of the item we want to delete, it first removes it from our applications data model and then calls a function to remove the object from the system, do not try to simply delete the object from `acs_objects` since this function exists to ensure that the objects are deleted in a safe way.

```
CREATE OR REPLACE FUNCTION todo_item__new (integer,varchar,text,char,
integer,date,varchar,integer)
RETURNS integer AS '
DECLARE
    p_item_id ALIAS FOR $1;
    p_title    ALIAS FOR $2;    -- default null
    p_description    ALIAS FOR $3; -- default null
    p_status      ALIAS FOR $4; -- default null
    p_creation_user ALIAS FOR $5; -- default null
    p_due_date    ALIAS FOR $6;
    p_creation_ip ALIAS FOR $7;    -- default null
    p_context_id ALIAS FOR $8;    -- default null

    v_id integer;
    v_type varchar;
BEGIN
```

```

v_type := "todo_item";

v_id := acs_object__new(
    p_item_id,
    v_type,
    now(),
    p_creation_user,
    p_creation_ip,
    p_context_id::Integer,
    true
);

insert into todo_item
    (item_id, title, description, status, due_date)
values
    (p_item_id, p_title, p_description, p_status, p_due_date);

return v_id;

END;
' LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION todo_item_delete (integer)
RETURNS VOID AS '
DECLARE
    p_item_id ALIAS FOR $1;
BEGIN
    delete from todo_item where item_id = p_item_id;
    PERFORM acs_object__delete(p_item_id);
END;
' LANGUAGE 'plpgsql';

```

**TRY IT!**

See appendix B for instructions on how to access your database.

---

## CHANGES TO THE PACKAGES PAGES:

We have modified our data model to use objects, but we need to make sure our application knows this and uses them, in order to do this we are going to make some small changes in our user visible pages.

---

### ADD AND EDIT WITH OBJECTS:

We will start the changes on the "todo-ae" page, the changes we need to make are to ensure that:

- ✓ When adding new items, we are adding them as objects
- ✓ When editing an item, we are updating the object
- ✓ When retrieving and displaying information, we are displaying the object info

---

### Changes Retrieving Information:

When retrieving information, either for editing or displaying the item's details, we need to change the "select\_query" block on the ad\_form definition, we are simple going to rewrite the query to be like this:

```
select todo.title,
       todo.description,
       to_char(todo.due_date, 'YYYY MM DD') as due_date,
       todo.status
from   todo_item todo,
       acs_objects obj
where  todo.item_id = :item_id
       and obj.object_id = todo.item_id
       and obj.creation_user = :user_id
       and obj.context_id = :package_id
```

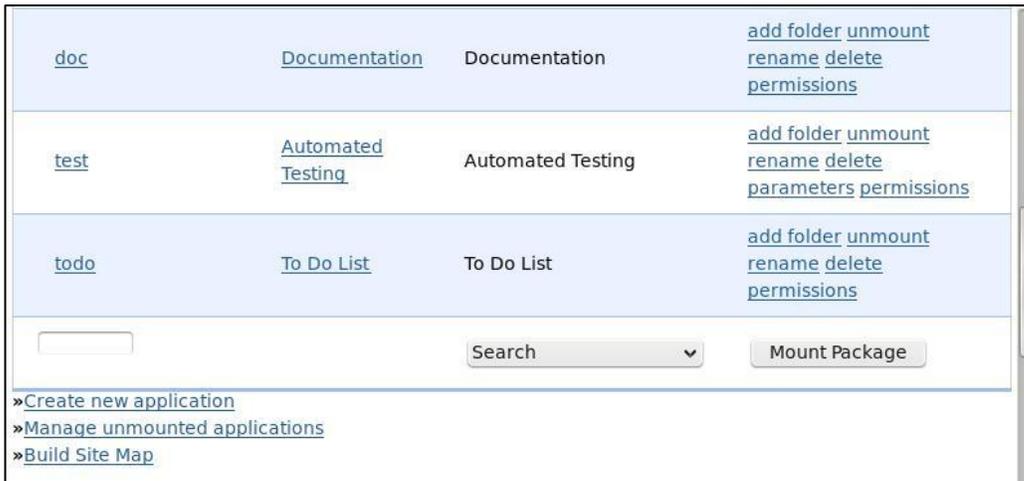
As you can see the main change here was getting some of the information out of the acs\_objects table, one of the fields included on the where clause, **context\_id** tells in what package instance was the item created and **creation\_user** field tells us who created this item, we use them to be sure we are displaying information that belongs to the correct instance and the appropriate user.

Remember when we used the **site map** to mount our first package? What we did there is to create a new **instance** of the package "todo", you can go there and mount as many instances you want, for example you could mount an instance named "todo-work" and another "todo-school" to keep track of your work and school activities on a separate instance, OpenACS will handle the logic behind creating the URL, permissions, users, etc.

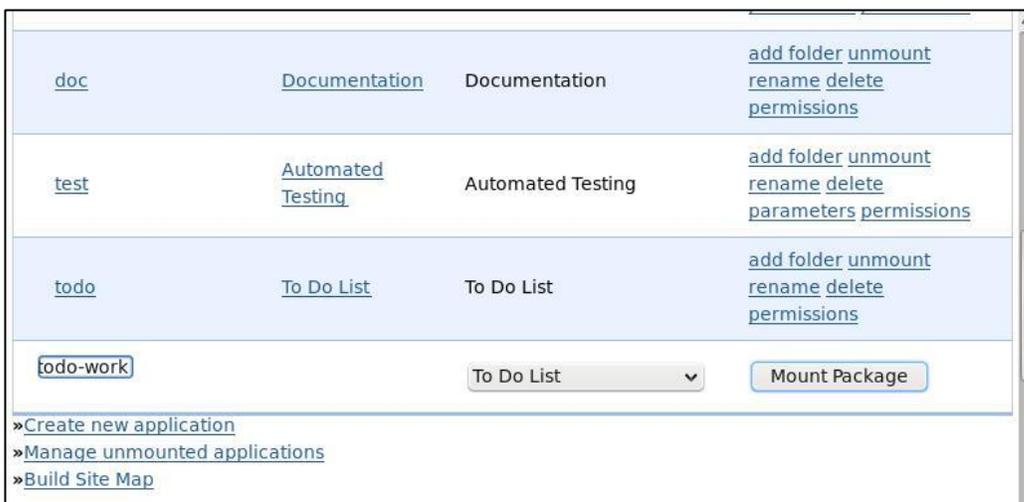
## WORKING WITH MULTIPLE INSTANCES:

To create multiple instances of the same package follow these steps:

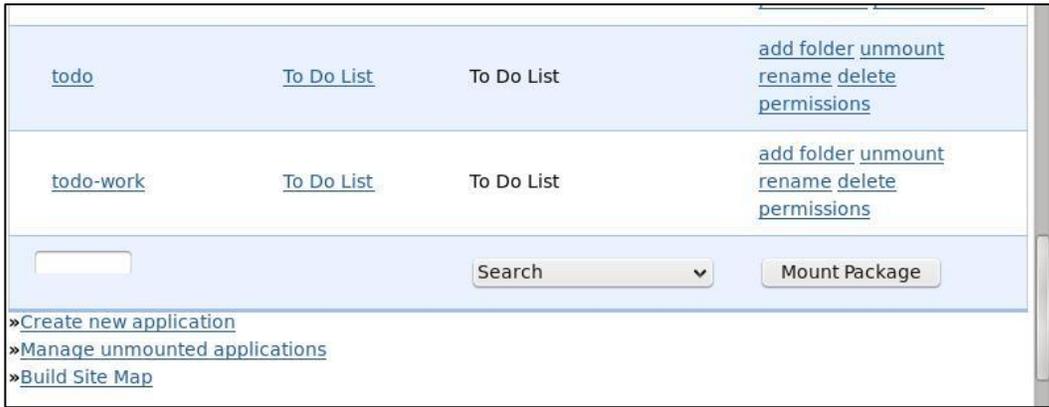
1. Head to your site's "Site Map", <http://localhost:8000/admin/site-map>



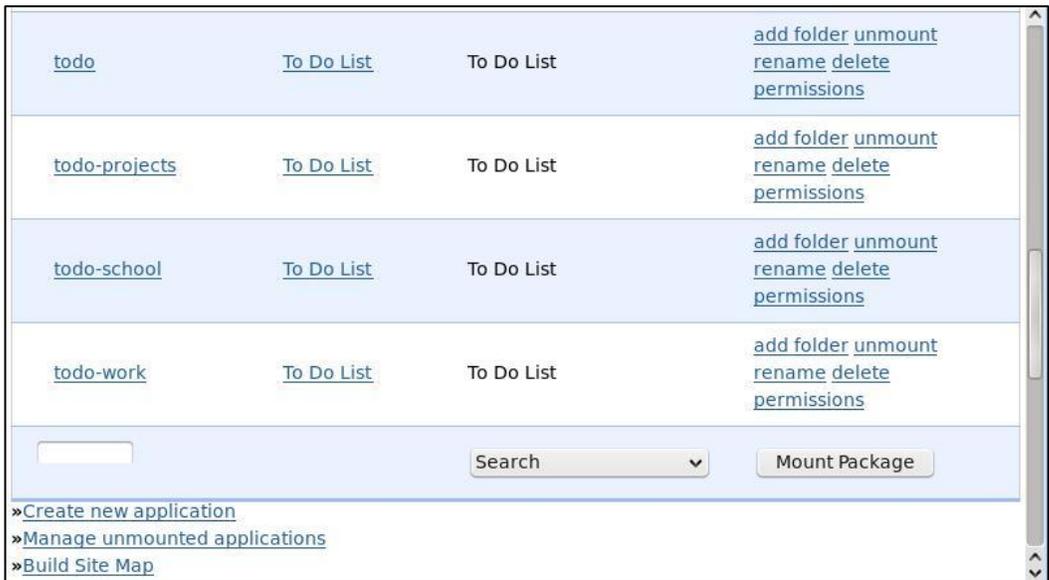
2. Following the same process we did when mounting our package for the first time, select "To Do List" from the dropdown box menu and give this new instance a new name, we called our new instance "todo-work"



3. Now click "Mount Package", the page will reload and you will have a new instance of our "To Do List" package mounted on, <http://localhost:8000/todo-work>



4. You can repeat this process and mount as many instances as you like.



5. If you click on the “Permissions” link for any instance, you can see and manage who has what privilege over this particular instance.

**Jaunty OpenACS** Welcome, Cesar Claveria | [1 Member Online](#) | [Your Account](#) | [Logout](#)

[Main Site](#) : [Permissions](#) : Permissions for To Do List

**Inherited Permissions**

- Cesar Claveria, admin
- The Public, read

**Direct Permissions**

Main Site Administrators, admin

Revoke Checked

[ [Grant Permission](#) | [Don't inherit Permissions from Main Site](#) ]

**Children**

none

[up to #acs-kernel.Main Site#]

[Install locales](#)

6. Click on “*Don’t inherit Permission’s from Main Site*” to remove the automatic permission’s this instance has obtained from the main site. In this case it will remove **the public’s** permission to read anything on this instance.

**Jaunty OpenACS** Welcome, Cesar Claveria | [1 Member Online](#) | [Your Account](#) | [Logout](#)

[Main Site](#) : [Permissions](#) : Permissions for To Do List

**Inherited Permissions**

- Cesar Claveria, admin

**Direct Permissions**

Main Site Administrators, admin

Revoke Checked

[ [Grant Permission](#) | [Inherit Permissions from Main Site](#) ]

**Children**

none

[up to #acs-kernel.Main Site#]

[Install locales](#)

## Changes Adding New Data:

---

The main changes here will be replacing our insert query for a call to the procedure that creates a new object of type **todo\_item**. All of these changes take place inside the "new\_data" block of the ad\_form definition.

```
-new_data {
  set context_id [ad_conn package_id]
  set new_object_id [ db_exec_plsql do_insert {
    select todo_item__new (
      :item_id,
      :title,
      :description,
      :status,
      :user_id,
      to_date(:due_date, 'YYYY MM DD HH24 MI SS'),
      :ip_address,
      :context_id
    );
  }]
}
```

As you can see we defined a "context\_id" variable, this is just for sakes of making the function call clearer on what parameter we are passing, our insert statement was replaced by a function call using [db\\_exec\\_plsql](#) (which is special for calling database functions).

## Changes editing an item:

---

The changes needed on our edit section are mainly to just deal with the information related to the object and to keep track of when this object was modified and by whom.

These changes go into the "edit\_data" block of the ad\_form definition.

```
-edit_data {
  # update the information on our table
  db_dml todo_item_update "
  update todo_item
  set title= :title,
      description = :description,
  status = :status,
      due_date = to_date(:due_date, 'YYYY MM DD HH24 MI SS')
  where item_id = :item_id"
```

```
# update the last modified information on the object
db_exec_plsql to_do_list_obj_item_object_update {
    select acs_object__update_last_modified(:item_id,:user_id,:ip_address)
}
}
```

To edit our item's information we just issued an *updated statement* on the database, for the object we used a call to the **acs\_object\_\_update\_last\_modified** function that will record when the last change was made, on what IP address and by which user.



```
ad_page_contract {
This page allows the users to add new items to their to do list or edit existing items.
}{
item_id:optional
{form_mode "edit" }
}

set page_title "Add/Edit Todo Item"
set user_id [ad_conn user_id]
set package_id [ad_conn package_id]
set ip_address [ad_conn peeraddr]

ad_form -name todo_item_form -export {user_id package_id} -mode $form_mode -form {
item_id:key
{title:text {label "Task Title"} }
{description:text(textarea),optional {label "Description"}}
{due_date:date(date) {label "Due Date: "} {format {MONTH DD YYYY}} }
{status:text(select) {label "Status"}
  {options { "Pending" "p"
             "Complete" "c"
             "Canceled" "x" }
  }}
}
}-select_query {
  select todo.title,
         todo.description,
         to_char(todo.due_date, 'YYYY MM DD') as due_date,
         todo.status
  from   todo_item todo,
         acs_objects obj
  where  todo.item_id = :item_id
         and obj.object_id = todo.item_id
         and obj.creation_user = :user_id
         and obj.context_id = :package_id
}-new_data {
  set context_id [ad_conn package_id]
```

```

set new_object_id [ db_exec_plsql do_insert {
  select todo_item__new (
    :item_id,
    :title,
    :description,
    :status,
    :user_id,
    to_date(:due_date, 'YYYY MM DD HH24 MI SS'),
    :ip_address,
    :context_id
  );
}]
} -edit_data {

# update the information on our table
db_dml todo_item_update "
  update todo_item
  set title= :title,
      description = :description,
  status = :status,
      due_date = to_date(:due_date, 'YYYY MM DD HH24 MI SS')
  where item_id = :item_id"

# update the last modified information on the object
db_exec_plsql to_do_list_obj_item_object_update {
  select acs_object__update_last_modified(:item_id,:user_id,:ip_address)
}

} -after_submit {
  ad_returnredirect index
  ad_script_abort
}

```

The ADP file does not need any changes at this point.

---

## INDEX PAGE WITH OBJECTS:

The index page that just presents a list of our *todo* items and allows us to add more, also have to be modified a little to make it work with our new data model.

### Changing the multirow:

---

Our first change will be in the multirow's sql query, we will need to add some information from the objects table, change the multirow's query to look like this:

```
select todo.item_id,
       todo.title,
       todo.due_date,
       to_char(todo.due_date, 'Month DD YYYY ') as due_date_pretty,
       obj.creation_date,
       to_char(obj.creation_date, 'Month DD YYYY ') as creation_date_pretty,
       todo.status
from todo_item todo, acs_objects obj
where obj.context_id = :package_id
      and obj.creation_user = :user_id
      and obj.object_id = todo.item_id
$orderby_clause
```

Notice how we pulled the creation date out of the **object\_id** and made sure the context of the objects we are displaying is inside our current package.

### Changing our orderby statement:

---

We will need to do a couple of adjustments to the orderby statements, nothing major, we will just make sure that we are ordering by the correct fields.

```
-orderby {
  title {orderby todo.title}
  due_date_pretty {orderby todo.due_date}
  status_text {orderby todo.status}
  creation_date_pretty {orderby obj.creation_date}
}
```

And those are all the changes we need on the user visible pages to make them work with objects system, but right now we just made sure we can retain the same functionality.

TRY IT!

<http://localhost:8000/todo/>

What can we gain from using the ACS Objects? We will explore that in the next section.

## SECTION 7

ON THIS SECTION YOU WILL LEARN:

- ✓ HOW TO INTEGRATE YOUR OBJECTS WITH OTHER OPENACS SERVICES
- ✓ HOW TO ADD THE SERVICE COMMENTS TO YOUR "TO DO" ITEMS AND OTHER OBJECTS
- ✓ HOW TO USE THE PERMISSIONS API

INTEGRATING WITH OTHER SERVICES:

One of the most noticeable benefit from using ACS objects is to be able to integrate your application with some of the services provided by OpenACS we will explore how to do it for a couple of them.

COMMENTS:

What we intend to do is to add the ability to store comments on a task, we could simply create our own data model and store our comments, but the real benefit of using OpenACS is taking advantage of all of the great modules and applications that already have been developed and tested with it. For our comments functionality we are going to use the "**General Comments**" service, that allows us to tie any ACS Object with comments and we'll do all of this in just a few lines of code.

But first, let's check if the "**General Comments**" package is already installed, to do this go to: <http://localhost:8000/acs-admin/install/> and under "Installed Packages" look for General Comments.

Installed Packages	
Type [ <b>Application</b>   <a href="#">Service</a> ]	
Package	Version
General Comments	5.2.0
Lars Blogger	2.4.1
Notifications	5.4.3
Search	5.4.3
To Do List	0.1d
To Do List Object	0.1d

If you do have it installed, continue to the next section “**Adding Comments to our Items**”, if not, the next steps will show you how to install it.

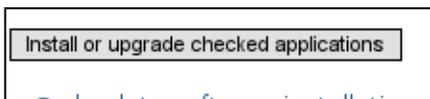
---

## INSTALLING GENERAL COMMENTS:

1. On the same page, click on "[Install from Repository](#)"
2. Now you should see a list of all of the available packages to download and install, look for the "General Comments" package and select it.

<input type="checkbox"/>	<a href="#">Forums</a>	Online discussion forums.
<input checked="" type="checkbox"/>	<a href="#">General Comments</a>	Service to provide comment entry and display on objects.
<input type="checkbox"/>	<a href="#">lmsld</a>	IMS LD integration with dotLRN.

3. Now click on the "Install or Upgrade Checked Applications" button near the bottom of the page.



4. A confirmation page will come up click on the "Install Above Package" to begin the installation process.

This is the package we are going to install.  
Please click the link below to begin installation.

Package	Version
General Comments	5.2.0

» [Install above package](#)

5. Wait for the packages to be installed and on the confirmation page click the link to restart your server.

Done installing packages.

» [Click here to restart your server now](#)

6. Depending on your set up you may need to restart your service manually.
  - ✓ On Ubuntu run this command on a terminal:
    - Sudo /etc/init.d/openacs restart
  - ✓ On Windows:
    - Go to the Start Menu
    - Look for the "win32-OpenACS" programs menu
    - Click on "Stop OpenACS" and then
    - Click on "Start OpenACS"

---

## ADDING COMMENTS TO OUR ITEMS:

We will display the comments when we are viewing just one single item on "display" mode, the comments will be hidden on "Edit" mode, we need to add link to add comments and display the content of the comments. Add the following code of block **after** the ad\_form definition on the file "**todo-ae.tcl**"

```
if { [string equal $form_mode "display"] } {  
  
    set comment_add_url "[general_comments_package_url]comment-add?[export_vars {  
    { object_id $item_id }  
    { return_url [util_get_current_url]}  
    }]"  
  
    set comments_html [general_comments_get_comments -print_content_p 1 $item_id  
    [util_get_current_url]]  
}
```

### Let's examine the code:

- ✓ First we are checking that this is only executed on "display" mode, so we will have the comments only when reviewing the item.
- ✓ Next we are building or "Add Comment" link, notice:
  - We use the **[general\_comments\_package\_url]** proc to get the URL to the general comments package, hardcoded URLs are not good idea on a system as flexible as OpenACS.
  - The use of **export\_vars** to pass a few variables that will be added to the link, those variables are the item\_id (as object\_id) and return\_url so we can return to this page after the comment is created.
- ✓ Finally we are getting **all** the comments on this item, already to be displayed by using the **general\_comments\_get\_comments** procedure.
  - We will display this on our adp.

We already got the information we want on the tcl file, now we need to display it on the adp.

Add this code to the "todo-ae.adp" file

```
<if @form_mode@ eq "display">
<a href="@comment_add_url@">Add a comment</a>
<p>
@comments_html;noquote@
</if>
```

Here we are again checking that we are on display mode, if we are, then we build and display the "Add a comment" link and we will display all of the comments related to this object.

Notice how we are getting the *comments\_html* variable content, @comments\_html;noquote@, the use of the **noquote** modifier is to prevent OpenACS from cleaning up and displaying the html markup instead of letting it pass to the browser to display as we intended. If you do not use it you will get something like:

```
<h4>My Comment</h4> Yes this is my comment!
```

Instead of a nice preformatted comments section.



TRY IT!

Try it now and your results should be similar to the next screenshot.

Task Title	Pick up Milk
Description	Remember the milk on my way home
Due Date:	June 10 2009
Status	Pending

[Add a comment](#)

**Also sugar**

Remember to also buy sugar.

-- [cesar claveria](#) on May 01, 2009 04:45 PM ([view details](#))

## PERMISSIONS:

This is probably one of the major benefits you may get out of using ACS Objects, easy integration with OpenACS permissions system. We are going to set up a little test case to see the permission system in action

First, **add a new user to your system**, you can do this by going to: <http://localhost:8000/acs-admin/users/user-add> and filling out the new users form, after hitting submit a new user will be created.

We will modify our code so only the user creating each item is able to read and modify them.

On the **todo-ae.tcl** file we will modify the "**new\_data**" section on the ad\_form, add this code after the object has been created, but before closing the new\_data block.

```
#The first command on this last block of code will remove all the permissions the object has
inherited from its "ancestors" (the package, the site, etc), the next command grants the "admin"
permission to the current user and finally, we allow the current user to be able to add
comments to this object.

#Remove all permissions from the object.
  permission::toggle_inherit -object_id $new_object_id

#Grant the permission to add comments to this user, on this object.
#Administrators will have this permission by default, but, we are making sure non-admin users
also get the correct permissions.
  permission::grant -party_id $user_id \
    -object_id $new_object_id \
    -privilege "general_comments_create"
```

Now add this permissions verification at the top of the page, just after the ad\_page\_contract.

```
if { [exists_and_not_null item_id] && [acs_object::object_p -id $item_id]} {
  if {[string equal $form_mode "edit"]} {
    permission::require_permission -object_id $item_id -privilege write
  } else {
    permission::require_permission -object_id $item_id -privilege read
  }
}
```

This will require the user to have the specified permissions before attempting to do anything, if the form is in display mode then the "read" permission is required, if the form is in edit mode then the "write" permission is required. The permissions system opens a world of possibilities.

Try it by trying to read or edit the objects with different users, you will see that when you are trying to read something to which you do not have permissions the system will detect it and stop you with an error message:

**yourdomain Network**

[Main Site](#) : To Do List Object

- *You don't have permission to read To Do Item 821.*



---

#### FINISHING THE TRANSITION:

There still a couple of pages we have not updated to use our new objects, the delete and updated status pages. This is how they look now with ACS Objects:



todo-delete.tcl

```
ad_page_contract {
  This page will delete, an item from the todo list package and then return the user
  to the specified return URL.
}{
  item_id
  return_url
}

permission::require_permission -object_id $item_id -privilege delete
set user_id [ad_conn user_id]
db_transaction {
  db_exec_plsql delete_item { select todo_item__delete ( :item_id ) }
}

ad_returnredirect $return_url
```

Notice how we have replaced the delete statement for a call to the **todo\_item\_delete** stored procedure, this way we ensure that both the item and the object are deleted correctly. We are also checking that the user has the correct permissions to *delete* this item.



```
ad_page_contract {
  This page will update an item's status from the todo list package and then return the user
  to the specified return url.
}{
  item_id
  new_status
  return_url
}

permission::require_permission -object_id $item_id -privilege write

set user_id [ad_conn user_id]
set status_code [todo::get_status_code $new_status]
db_transaction {
  db_dml update_status "update todo_item
                        set status = :status_code
                        where item_id = :item_id"
}

ad_returnredirect $return_url
```

This page has not changed much, we are just making sure the user has the required permissions to carry out the operation. We waited to update these pages so we could take advantage of the user permissions checking.

And that is all for objects for now, they are a rich tool and their integration with the other OpenACS services allows you to add a great deal of functionality in an easy and organized way.

## CONCLUSIONS

Still we need at OpenACS quick tutorial for important areas such as xql, libraries, advanced examples of ad\_form & list builder, usage of the content repository, more advanced ACS Objects examples, etc, so hopefully more people will be interested to develop new quick tutorials for OpenACS and its features.

This tutorial has been designed and organized by Cesar Clavería & Rocael Hernández. Mostly written by Cesar Clavería, reviewed by Rocael Hernández and many others have helped to review and improve it, tested with many OpenACS beginners.

This works has been done with help of the .LRN Consortium.

***Getting Started with OpenACS:  
your quick guide to a powerful framework.***

***César Clavería – Rocael Hernández***

***2009***

## APPENDIX A

### OPENACS ON WINDOWS:

Thanks to the flexibility of OpenACS and the software it is built upon it is possible to run an OpenACS system on different environments, including Microsoft Windows © at the moment of this writing the Windows installer works with Windows XP and Vista, both on the 32 bits version.

Next is a table detailing where to find the different directories we discussed on the tutorial, this paths are valid if you used the windows installer.

Directory	Ubuntu	Windows
<b>Main OpenACS installation</b>	<code>/usr/share/openacs</code>	<code>C:\aolserver\servers\openacs</code>
<b>OpenACS packages directory</b>	<code>/usr/share/openacs/packages</code>	<code>C:\aolserver\servers\openacs\packages</code>
<b>Our first “To Do” package</b>	<code>/usr/share/openacs/packages/todo</code>	<code>C:\aolserver\servers\openacs\packages\todo</code>
<b>OpenACS configuration file</b>	<code>/etc/openacs/config.tcl</code>	<code>C:\aolserver\servers\openacs\etc\config.tcl</code>
<b>OpenACS log file</b>	<code>/usr/share/openacs/log/error.log</code>	<code>C:\aolserver\servers\openacs\log\error.log</code>

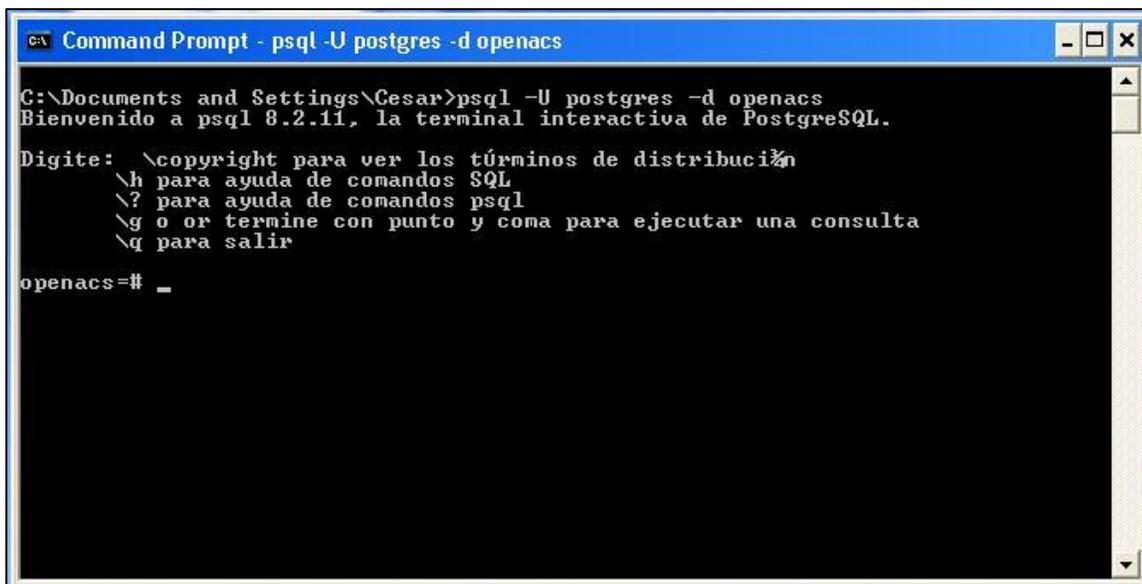
## APPENDIX B

### CONNECTING TO POSTGRESQL ON WINDOWS AND UBUNTU:

There are many ways to connect to your OpenACS database on both Windows and Linux, we will deal here exclusively with how to connect to the database using the built-in **psql** command line utility always included on a Postgresql installation.

#### CONNECTING ON WINDOWS:

1. Open a command prompt window.
  - ✓ Go to, Start → Run → cmd.exe and press enter.
2. Connect as the postgres user by entering the command:
  - ✓ **psql -U postgres -d openacs**
    - If you are asked for a password the default password for the postgres user when using the win32 installer is "qwe123"
3. Now you can perform any sql command on this console.



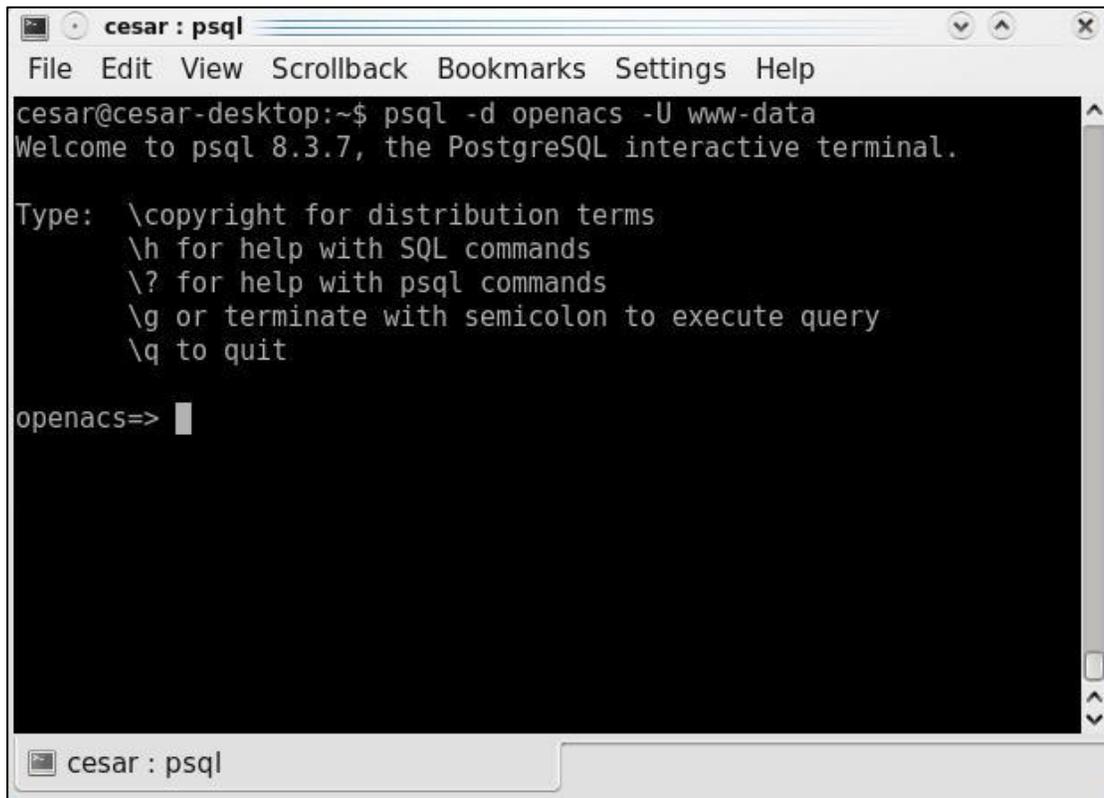
```
Command Prompt - psql -U postgres -d openacs
C:\Documents and Settings\Cesar>psql -U postgres -d openacs
Bienvenido a psql 8.2.11, la terminal interactiva de PostgreSQL.

Digite: \copyright para ver los t rminos de distribuci n
        \h para ayuda de comandos SQL
        \? para ayuda de comandos psql
        \g o or termine con punto y coma para ejecutar una consulta
        \q para salir

openacs=# _
```

## CONNECTING ON UBUNTU:

1. Open up a terminal application
  1. On Gnome: Applications → Accessories → Terminal
  2. On KDE: K Menu → Applications → System → Terminal
2. Enter the command:
  1. **psql -d openacs -U www-data**
3. You should be able to enter any sql command.



```
cesar : psql
File Edit View Scrollback Bookmarks Settings Help
cesar@cesar-desktop:~$ psql -d openacs -U www-data
Welcome to psql 8.3.7, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help with psql commands
      \g or terminate with semicolon to execute query
      \q to quit

openacs=> |
```