

XOTcl for OpenACS

An Introduction to XOTcl and the basic infrastructure of xotcl-core

Gustaf Neumann, Stefan Sobernig

Institute for Information Systems and New Media

Vienna University of Economics and Business Administration

Table of Contents

- . Part 1: XOTcl and OO Introduction
 - . Objects, Classes, Meta-Classes, Mixin Classes
 - . Attribute Slots
- . Part 2: XOTcl and the OpenACS Type System
 - . Database Access Layer
 - . OO Abstractions, API
 - . Content Repository
- . Part 3: Creating Applications with xotcl-core
 - . Simple Note Example

XOTcl - Short Introduction

XOTcl (www.xotcl.org) is one of several OO extensions for Tcl.

Important properties

- . efficient (implemented in C)
- . Similar to CLOS object system
- . Relations between objects and classes (and between classes) are dynamically changeable relations
- . All data is stored in object
- . Class are Objects
- . Classes of a Class are Meta-Classes

XOTcl Classes

Stack: the classical example for classes

```
#
# Create a stack class
#
Class Stack

Stack instproc init {} {
    # Constructor
    my instvar things
    set things ""
}

Stack instproc push {thing} {
    my instvar things
    set things [concat [list $thing] $things]
    return $thing
}

Stack instproc pop {} {
    my instvar things
    set top [lindex $things 0]
    set things [lrange $things 1 end]
    return $top
}
```

Using Stack in a script

Using the class "Stack" in a script

```
% package req XOTcl
% namespace import xotcl::*
% source stack.xotcl

# Create Object s1 of class Stack
% Stack s1
::s1
% s1 push a
a
% s1 push b
b
% s1 push c
c
% s1 pop
c
% s1 pop
b
# Delete object s1
s1 destroy
```

Mixin Classes

XOTcl supports in addition to classical class hierarchies Mixin classes to implement orthogonal aspects

- . Per-Object Mixins: to modify the behavior of a single object
- . Per-Class Mixins: to modify the behavior of all instances of a class

Typical examples: loggers, persistence, renderers,

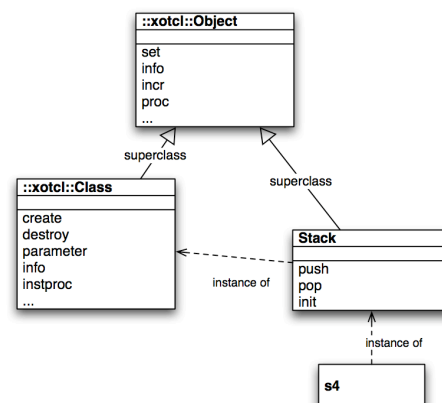
Mixin Class Safety

Previous Stack example does not check for underruns. Implement "Safety" as separate class, which can be used as a mixin.

```
#
# Create a safety class
#
Class Safety
Safety instproc init {} {
    my set count 0
    next
}
Safety instproc push {thing} {
    my incr count
    next
}

Safety instproc pop {} {
    if {[my set count] == 0} then { error "Stack empty!" }
    my incr count -1
    next
}
```

XOTcl Base and Application Classes



Classes and Meta-Classes

A meta-class can be used to define different kind of classes (e.g. persistent classes)

```
# Create a generic object
Object o1

# Create a different kind of object: create a new class
Class Person
Person p1

# Create a different kind of class: create a new meta-class
Class PersistentClass -superclass Class

# Create a persistent class
PersistentClass Page

# Create an instance of that class
Page p2
```

Mixin Class Safety

Important:

- . "Safety" overloads certain methods of "Stack"
- . Method chaining via "next"
- . Ability to perform arbitrary code for the overloaded methods before and after the shadowed method call

Same mechanism for the

- . intrinsic class hierarchy
- . mixins
- . filters (not covered here)

Use Safety as per-object mixin

We can use Safety as a per-object mixin

```
% Stack s2 -mixin Safety
::s2
% s2 push a
a
% s2 pop
a
% s2 pop
Stack empty!
```

The mixin can be added/removed to existing objects at any time

Use Safety as per-class mixin

The same class Safety can be used as per-class mixin.

```
#
# Create a safe stack class by using Stack and mixin
# Safety
#
Class SafeStack -superclass Stack -instmixin Safety

SafeStack s3
```

All instances of SafeStack are now safe.

Per-Class mixins are transitive.

Mixins are an instrument of composition

Stack for integers

Methods can be defined per-class (instproc) or per-object (proc).

Define a single stack for integers:

```
#
# Create a stack with a object-specific method
# to check the type of entries
#
# s4 is a stack of integer

Stack s4
s4 proc push {value} {
    if {[string is integer $value]} {
        error "value $value is not an integer"
    } next
}
```

Meta-class, Class Object

Create a simple object: create an instance of ::xotcl::Object as superclass

```
Object o1
# ... is a short form for ...
Object create o1
```

Create a class: create an instance of ::xotcl::Class

```
Class Person
# ... is a short form for ...
Class create Person
# ... or ...
Class create Person -superclass Object
```

Person can now be used to define specialized objects (Persons)

Methods for classes

Since classes are Objects, one can define class-specific methods (similar to static methods in C++/Java) with the same method as object-specific methods (via "proc")

```
Class Stack
# ...
Stack proc available_stacks {} {
    return [llength [my info instances]]
}

Stack s1
Stack s2

puts [Stack available_stacks]
```

XOTcl Object System

- . All classes have a common root class (::xotcl::Object)
- . The behavior (methods) of the common root class is inherited to all other classes
- . Methods can be added at any time using the methods "proc" and "instproc"
- . XOTcl classes are as well Objects, instances of a meta-class ::xotcl::Class
- . As a class provides methods to objects, a meta-class provides methods to classes

Slots and Attribute Management

Slots are meta-objects that manage property-changes of objects. A property is either an attribute or a role in an relation. In a nutshell, a slot has among other attributes:

- . a *name* (which it used to access it),
- . a *domain* (object or class on which it can be used) , and
- . can be *multivalued* or not.

We distinguish between *system slots* (predefined slots like `class`, `superclass`, `mixin`, `instmixin`, `filter`, `instfilter`) and *attribute slots* (e.g. attributes of classes).

Attribute Slots

Attribute slots are used to manage the setting and querying of instance variables. We define now a person with three attributes `name`, `salary` and `projects`.

```
Class Person -slots {  
  Attribute name  
  Attribute salary -default 0  
  Attribute projects -default {} -multivalued true  
}
```

Attributes might have a default value or they might be multivalued. When an instance of class `Person` is created, the slot names can be used for specifying values for the slots.

```
Person p1 -name "Joe"
```

Object `p1` has three instance variables, namely `name`, `salary` and `projects`.

Multivalued slots

Since slot `projects` is multivalued, we can add a value to the list of values the `add` subcommand.

```
Project project1 \  
  -name XOTcl \  
  -description "A highly flexible OO scripting language"  
  
p1 projects add ::project1  
p1 projects add some-other-value
```

The value of the instance variable `project` of Person `p1` is now the list `{some-other-value ::project1}`.

More Advanced Slot Options

Slots in XOTcl are extensible:

- . Slots manage properties like "class", "superclass", "mixin"
- . Slots provide methods for setter and getter of values
- . Slots carry meta-data about attributes
- . Slots are classes, one can therefore define different kind of slots (e.g. slots with corresponding attributes in the database)

More details in the [XOTcl Tutorial](#)

Summary

Objects

- . Carry data

Classes

- . Manage objects (lifecycle management)
- . Provide methods for objects

Slots

- . Provide meta-data for properties (mostly attributes)
- . Provide methods for accessing attributes

Questions?



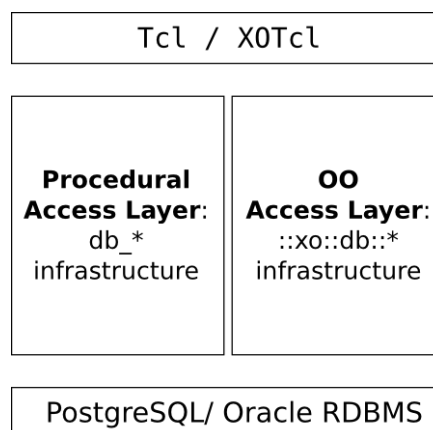
XOTcl and the OpenACS Type System

OpenACS has its own Type Sytem

- . acs-objects
- . acs-object-types
- . acs-attributes

Linkage between XOTcl Object System to the OpenACS Type System provided through xotcl-core

Database Access Layers

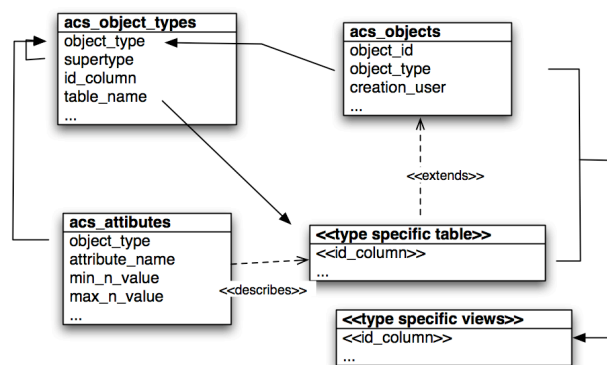


xotcl-core Object/Relational Access Layer

Facilities:

- . Object proxies for Oracle/ PostgreSQL stored procedures and modules (not covered here)
- . XOTcl representation of OpenACS Object System ("Split object pattern")
- . Use existintg acs-object-types as XOTcl classes
- . Create oacs-object-types/acs-objects by creating XOTcl classes / objects
- . Additional support for the OpenACS Content Repository

Simplified OpenACS Meta Model

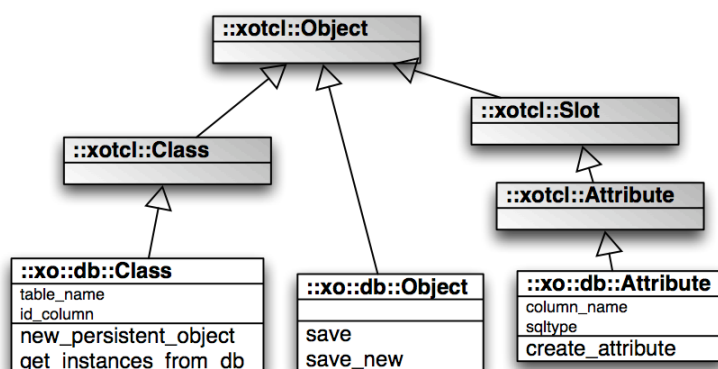


Similarities between OpenACS Meta Model and XOTcl Concepts

- . **acs_object_types**: defines specialization - similar to XOTcl classes
- . **acs_objects**: defines common attributes - similar to `::xotcl::Object`
- . **acs_attributes**: defines properties of attributes - similar to attribute slots

DB-Layer is much less flexible then XOTcl:
less support for dynamic operations (adding attributes), no support for re-classing, mixins, ...

XOTcl Classes for the Persistent Object Layer

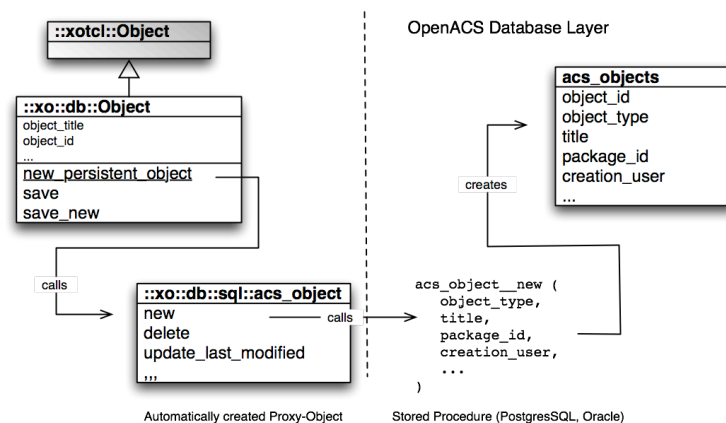


Creating ACS Objects

Approach:

- . Create an instance of `::xotcl::db::Object`
- . Using a specialized method (`new_persistent_object`), to distinguish between creating only XOTcl object and as well acs-objects in the database
- . acs-objects are created in general via stored procedures in SQL
- . xotcl-core auto-generates for all stored-procedures proxy-objects from the db-system catalog

Creation of an ACS Object



Interface for `acs_object__new`

```
::xo::db::sql::acs_object__new [ -dbn dbn ] ...
```

Defined in `packages/xotcl-core/tcl/05-db-procs.tcl`

Switches:

- dbn (optional)
- object_id (optional)
- object_type (defaults to "acs_object") (optional)
- creation_date (defaults to "now()") (optional)
- creation_user (optional)
- creation_ip (optional)
- context_id (optional)
- security_inherit_p (defaults to "t") (optional)
- title (optional)
- package_id (optional)

Create an ACS Object from XOTcl

```
#####
#
# 1) Create new ACS Objects, destroy it in memory,
#    load it from the database, delete it in the database.
#
.. Create a plain new ACS object just for demo purposes.
.. The ACS object is created with a new object id.

>> set o [::xo::db::Object new_persistent_object]
= ::7845

.. Show the contents of object ::7845 by serializing it:

>> ::7845 serialize
= ::xo::db::Object create ::7845 -noinit \
  -set object_title {Object 7845} \
  -set object_id 7845
```


Check Existence of an XOTcl Object, Destroy it

```
# In the next steps, we (a) get the object_id of the newly
# created ACS object, (b) destroy the XOTcl object (the ACS
# object is still in the database, (c) we recreate the
# XOTcl object from the database, and (d) delete it in the
# database.

.. Step (a)
>> set o_id [::7845 object_id]
= 7845

#
# Delete object from memory: <object> destroy
# Check, if an XOTcl object exists: ::xotcl::Object isobject <obj>
#
>> ::xotcl::Object isobject ::7845
= 1

.. Step (b)
>> ::7845 destroy

>> ::xotcl::Object isobject ::7845
= 0
```

Load Object from Database

```
#
# Load an object from the database:
# ::xo::db::Class get_instance_from_db -id <id>
#
.. Step (c)
>> set o [::xo::db::Class get_instance_from_db -id 7845]
= ::7845

>> ::xotcl::Object isobject ::7845
= 1

.. Now, we have recreated the same object as before:

>> ::7845 serialize
= ::xo::db::Object create ::7845 -noinit \
  -set object_title {Object 7845} \
  -set object_id 7845
```

Delete ACS Object via OO Interface

```
#
# Check, if an ACS object exists in the database:
#   ::xo::db::Class exists_in_db -id <id>
#
# Delete object from memory and database:
#   <object> delete
#
>> ::xo::db::Class exists_in_db -id 7845
= 1

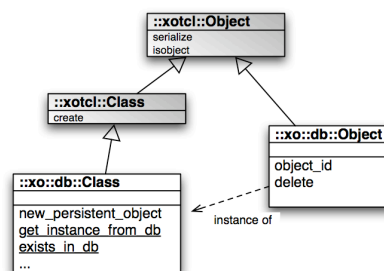
.. Step (d)
>> ::7845 delete

# Now, we have deleted the ACS Object and the XOTcl object:

>> ::xo::db::Class exists_in_db -id 7845
= 0

>> ::xotcl::Object isobject ::7845
= 0
```

Methods used so far



Note, that methods from the meta-class (create, new_persistent_object) are used on instances of the meta-class

Creating ACS Object Types

Approach:

- . Create a new XOTcl class via meta-class `::xo::db::Class`
- . Define new top-level object types as subclass of `::xo::db::Object`
- . If not provided, xotcl-core generates from the class name and namespace the table-name and id_column
- . xotcl-core creates all tables and necessary entries in acs-object-types, acs-attributes etc. from the XOTcl class definition

Create a new ACS Object Type

```
#####
#
# 2) Create new ACS Object Types, ACS Attributes and
#     SQL Tables from XOTcl Classes with slot definitions.
#
.. Create a new ACS Object type and an XOTcl class named ::demo::Person.

.. Does the ACS Object type ::demo::Person exist in the database?
>> ::xo::db::Class object_type_exists_in_db -object_type ::demo::Person
= 0

# We create a new XOTcl Class '::demo::Person'.
# By defining this class, the database layer takes care
# of creating the ACS Object Type and the necessary table via SQL.

.. The persistent attributes (stored in the database) are defined
.. as slots of type ::xo::db::Attribute.

>>
::xo::db::Class create ::demo::Person \
  -superclass ::xo::db::Object \
  -slots {
    ::xo::db::Attribute create name -column_name pname
    ::xo::db::Attribute create age -default 0 -datatype integer
    ::xo::db::Attribute create projects -default {} -multivalued true
  }

= ::demo::Person
```

The Created Artefacts

```
# If the ACS Object Type and the ACS Attributes would be
# already defined in the database, the class definition above
# would be a no-op operation.

# Now, the ACS Object Type exists in the database

>> ::xo::db::Class object_type_exists_in_db -object_type ::demo::Person
= 1

# The XOTcl class definition created automatically the
# following table for storing instances:

CREATE TABLE demo_person (
    age integer DEFAULT '0' ,
    pname text ,
    projects text DEFAULT '' ,
    person_id integer REFERENCES acs_objects(object_id) ON DELETE CASCADE
    CONSTRAINT demo_person_person_id_pk PRIMARY KEY
)

.. SQL attribute slot names:
>> ::demo::Person array names db_slot
= name age projects person_id
```

Created Methods

```
# The XOTcl class definition created as well a 'save' and
# an 'insert' method (latter omitted here):

.. Created 'save' method:

::demo::Person instproc save {} {
    db_transaction {
        next
        my instvar object_id name age projects
        db_dml dbqd..update_demo_person {
            UPDATE demo_person
            SET pname = :name,age = :age,projects = :projects
            WHERE person_id = :object_id
        }
    }
}
```

Create Instance of New Object Type

```
#
# Create a new instance of ::demo::Person with name 'Gustaf'
#
# The method 'new_persistent_object' of a database class (instance of ::xo::db::Cl
# creates an ACS Object with a fresh id in the database and
# creates as well an XOTcl object in memory

>> set p [::demo::Person new_persistent_object -name Gustaf -age 105]
= ::7846

.. check, if object ::7846 exists in memory
>> ::xotcl::Object isobject ::7846
= 1

.. check, if object ::7846 exists in the database
>> ::xo::db::Class exists_in_db -id 7846
= 1

.. Show the contents of object ::7846 (using serialize)

>> ::7846 serialize
= ::demo::Person create ::7846 -noinit \
  -set object_title {Person 7846} \
  -set name Gustaf \
  -set age 105 \
  -set projects {} \
  -set person_id 7846 \
  -set object_id 7846
```

Modify Object, Save and Fetch

```
# modify some attributes of the XOTcl object
>> ::7846 incr age
= 106

# save the modified object data in the database
>> ::7846 save

# deleting xotcl object ::7846 in memory
>> $p destroy

# check, if object ::7846 exists in the database
>> ::xo::db::Class exists_in_db -id 7846
= 1

# fetch person again from database:
>> set p [::xo::db::Class get_instance_from_db -id 7846]
= ::7846

# serialized content
::demo::Person create ::7846 -noinit \
  -set object_title {Person 7846} \
  -set name Gustaf \
  -set age 106 \
  -set projects {} \
  -set object_id 7846 \
  -set person_id 7846
```

Creating a Subclass

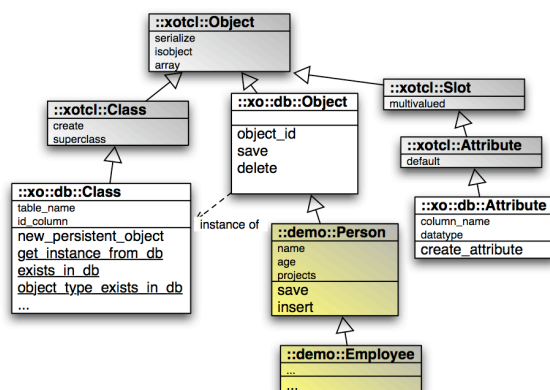
```
# Now, we create a subclass of ::demo::Person called ::demo::Employee
# which has a few more attributes. Again, we define an XOTcl class
# ::demo::Employee which creates the ACS Object Type, the ACS
# attributes and the table, if necessary.

>> ::xo::db::Class create ::demo::Employee \
    -superclass ::demo::Person \
    -table_name demo_employee \
    -id_column employee_id \
    -slots {
        ::xo::db::Attribute create salary -datatype integer
        ::xo::db::Attribute create dept_nr -datatype integer -default "0"
    }
= ::demo::Employee

# The XOTcl class definition created automatically the
# following table for storing instances:

CREATE TABLE demo_employee (
    dept_nr integer DEFAULT '0' ,
    salary integer ,
    employee_id integer REFERENCES demo_person(person_id) ON DELETE CASCADE
    CONSTRAINT demo_employee_employee_id_pk PRIMARY KEY
)
```

Methods used so far



XOTcl Classes from OpenACS Meta-data

```
#####
# 3) Create XOTcl classes from existing ACS Object Types
#    and ACS Attributes based on the definitions in the
#    database

>> set c [::xo::db::Class get_class_from_db -object_type party]
= ::xo::db::party

.. XOTcl class ::xo::db::party created (superclass ::xo::db::Object)
.. SQL attributes:
>> ::xo::db::party array names db_slot
= email party_id url

>> set c [::xo::db::Class get_class_from_db -object_type person]
= ::xo::db::person

.. XOTcl class ::xo::db::person created (superclass ::xo::db::party)
.. SQL attributes:
>> ::xo::db::person array names db_slot
= last_name first_names person_id
```

Query Interface

<instance of ::xo::db::Class> get_instances_from_db ...

Returns a set (ordered composite) of the answer tuples of an 'instance_select_query' with the same attributes. Note, that the returned objects might be partially instantiated.

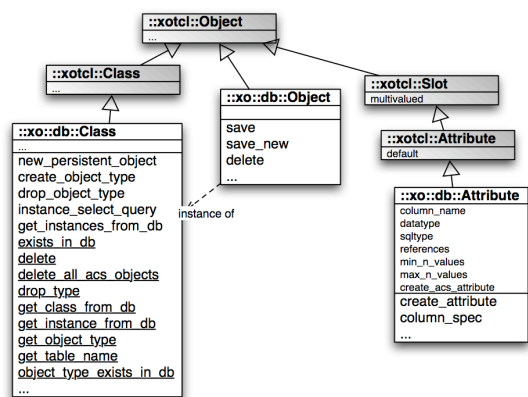
Switches:

- select_attributes (optional)
- from_clause (optional)
- where_clause (optional)
- orderby (optional)
- page_size (defaults to "20") (optional)
- page_number (optional)

Returns:

ordered composite

Summary of the xo::db Interface

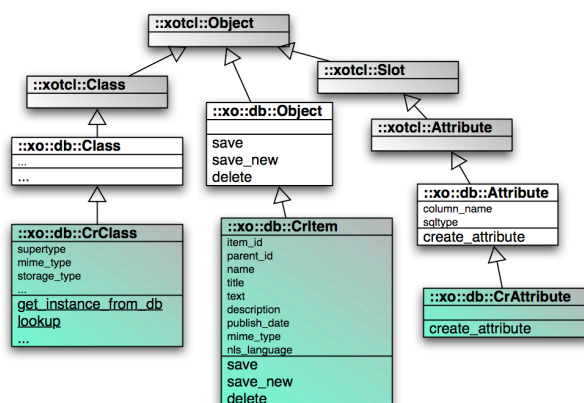


Interface to content repository

OpenACS Content Repository (CR):

- . Distinction between content-item and content-revision
- . Updates are revisioned
- . Most attributes are stored in the revisions
- . Content-items and content-revisions are acs-objects
- . Content-items are named entries, unique per parent_id (e.g. folder)
- . Extend attributes via subtyping/subclassing
- . Single XOTcl layer, accepting either item_ids or revision_ids

Overview of the XOTcl CR Interface



Creating Application specific Subclasses of the CR

```
#####
# Create new application classes by sub-typing the
# Content Repository, adding additional attributes
#
# We create a subclass of ::xo::db::CrItem called ::demo::Page
# which has a few more attributes. Actually, this class is very
# similar to ::xowiki::Page. Again, we define an XOTcl class
# ::demo::Page which creates the ACS Object Type, the ACS
# attributes and the table, if necessary.
```

```
>> ::xo::db::CrClass create Page \
    -superclass ::xo::db::CrItem \
    -pretty_name "demo Page" \
    -mime_type text/html \
    -slots {
        ::xo::db::CrAttribute create creator \
            -column_name creator_string
    }

= ::demo::Page
```

Create Instance, Save, Delete, Fetch

```
# create a page object in memory
>> set i [::demo::Page new \
    -name "page0" \
    -title "Joke of the Month"
    -creator "GN" -text "Three cannibals meet in a NYC subway station..." ]
= ::xotcl::__#j

# save as a new item under default parent_id (-100), allocates fresh item_id
>> $i save_new
= 7855

>> set item_id [$i item_id]
= 7855

# destroy object in memory
>> $i destroy

# fetch item per item_id from the database
>> ::demo::Page get_instance_from_db -item_id 7855
= ::7855
```

Fetch Item by Name

```
# Lookup page from CR by name
>> ::xo::db::Class lookup -name page0
= 7855

# fetch item per item_id from the database
>> ::demo::Page get_instance_from_db -item_id 7855
= ::7855

# modify the object
>> ::7855 set title "Kilroy was here"

# save the object with a new revision
>> ::7855 save
```

Questions?



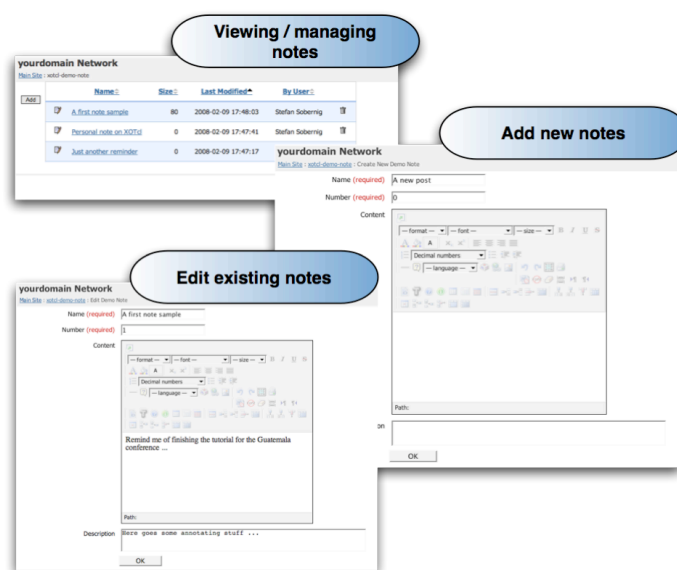
Creating Applications with XOTcl Core



Overview (1)

- Developing for OpenACS means creating OpenACS packages, being discrete modules in the OpenACS architecture.
- This tutorial is build around a little story: We want to create a note-keeping application call XOTcl Demo Note.
- There is a ready-made demo package available, for those who want to kick-start their own development (see below for reference).
- Use advanced OpenACS features (Object System, Content Repository) right from the beginning, facilitated by the XOTcl Core infrastructure.
- We aimed at sticking close to the message conveyed by the classic OpenACS development tutorial and show the XOTcl way of realising it ...

Overview (2)



Prerequisites

To get started with our demo package, you will need the following:

- . A running OpenACS installation that provides the XOTcl Core (package: xotcl-core). You find step-by-step instructions for installing XOTcl Core at the OpenACS wiki.
- . The examples files, or rather, the distribution of the underlying demo package, i.e. xotcl-demo-note. We host a copy at ...
- . An authoring environment to create and modify the script files involved.

Assumptions

For the scope of this tutorial, we follow a couple of assumptions. Be aware of these, provided that you apply lessons taken from here in a different context ...

- . URI of your OpenACS installation: **`http://localhost:8000`**
- . Package key of the demo package to be created: **`xotcl-demo-note`**
- . Whenever we refer to the "packages" directory of your OpenACS installation, we mean: `<path_to_your_OACS>/packages/`

Introduction in three Units

Guide to the first xotcl-core application in three Units:

- . Unit 1 / Package management:
 - . Creating and initialising a new OpenACS package, using the ACS Package Manager (APM)
 - . Understanding the file system and logical structure of OpenACS packages
 - . XOTcl Core support for handling and using OpenACS application packages
- . Unit 2 / Data Model
- . Unit 3 / Web User Interface:
 - . Using request and other context information
 - . Creating simple user dialogs
 - . Application's user interface (tables, lists, ...) through xotcl-core widgets

Unit 1 / OpenACS Packages

- . APM (ACS Package Manager) packages have various representations:
 - . File-system artifact
 - . Data base artifact (in the sense of ACS objects)
 - . XOTcl (in-memory) objects
- . Kinds of APM packages:
 - . "Application"
 - . "Singleton"

Unit 1 / Creating OpenACS Application Packages

<http://localhost:8000/acs-admin/apm> > Create New Package

Main Site : Site-Wide Administration : Package Manager : Add a New Package

Select a package key for your package. This is a unique short and lower-case identifier for your package. For the address book package or photo-album package will be used.

Important: A package key

Package Key:

Select a short, human-readable name for your package, e.g., "Address Book" or "Photo Album."

Package Name:

Please indicate the plural form of the package name, e.g. the plural form of "Bboard" is "Bboards."

Package Plural:

Indicate whether this package is an application or a service. Applications are software intended for OpenACS to provide a workflow.

Important: The package type

Package Type: ☒ Application ☐ Service

OpenACS Core? ☐ Is your package part of the OpenACS Core that forms the set of packages initially installed? If you'd leave this box unchecked.

Singleton? ☐ Is your package a singleton package? Singleton packages can have at most one instance; attempt to create a second instance will fail. Non-singleton packages can have multiple instances, so you can create a new instance at any time.

Auto-mount URI:

The URI (name) under the main site where the package will be mounted upon install.

Optionally: Auto-mount feature

Package URL:

Pick a canonical URL for your package. This should be a URL where the package can be downloaded.

Initial Version:

Select an initial version number for the package. By convention, this is 0.1.0 if you are just starting a new package. The version number should be in the form of **number minor number** with an optional patch level.

Mandatory: Provide an initial version tag

Version URI:

Type a brief, one-sentence-or-less summary of the functionality of your package. In general, this summary should begin with a capital letter and end with a period.

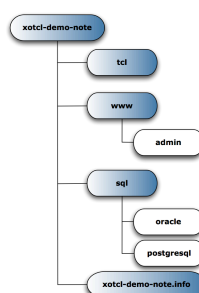
Summary:

61 von 95

13.02.2008 8:21 Uhr

Unit 1 / File Structure for APM Packages

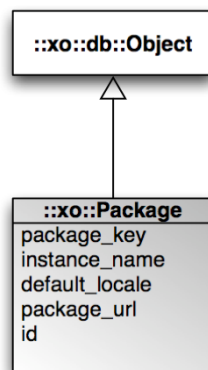
- Once the package manifesto is updated, the package will be created in the file system
- In our example, the following structure is created:



62 von 95

13.02.2008 8:21 Uhr

Unit 1 / Packages as XOTcl Objects



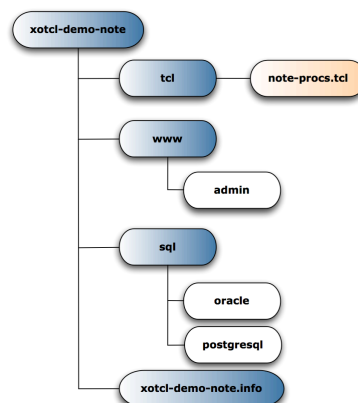
Unit 1 / Packages as XOTcl Objects

- Library script: `xotcl-core/tcl/06-package-procs.tcl`
 - Package manager: `::xo::PackageMgr`
 - Base package: `::xo::Package`
- `::xo::Package` realises various features, beyond a mere object representation of APM packages
 - It may act as message redirector
 - It acts as **context resolver** (site node > package_id)
 - It amends and integrates legacy OpenACS facilities, e.g. ad page contract
 - Provides access to per-instance parameters (APM parameters)
 - In more advanced scenarios, it is point of reference for providing per-instance folders ...

Unit 1 / Manage Package Instances

- . In our demo package, we want to define a new kind of package manager:
::demo::Package
 - . **Step 1:** Define a new class of package by defining an XOTcl class
 - . **Step 2:** Assign initialisation behaviour to instances of this new package type
- . Create a script file called "note-procs.tcl" in xotcl-demo-note/tcl/ and place the following code snippets there.

Unit 1 / Library File note-procs.tcl



Unit 2 / Define a new Package Type

```
# Define classes in the Tcl namespace ::demo in file note-procs.tcl

namespace eval ::demo {
  # Define a sub-class of ::xo::Package
  # and provide some meta-data (package-key, pretty_name ...)

  ::xo::PackageMgr create Package \
    -superclass ::xo::Package \
    -package_key "xotcl-demo-note" \
    -pretty_name "XOTcl Demo Note Package" \
    -parameter {{folder_id 0}}
  ...
}
```

Unit 1 / Package manager

Step 2: Define additional, type-specific initialisation behaviour (constructor)

- . Type-specific behaviour: We want to store all notes (instances of ::demo::Note) for each package instance in an own OpenACS Content Repository folder.
- . Conceptually, each package instance (bound to a site node) is associated to a single folder as item container.

Unit 1 / Package Constructor

```
# We are still in the Tcl namespace ::demo in file note-procs.tcl

...

# The folder_id is an parameter of the Package.
# Each time we initialize our package, we want to have the
# folder id ready. So we extend the basic ::xo::Package class,
# provide an additional parameter folder_id and create the folder on
# the fly when necessary.

Package instproc init {} {
    next
    my folder_id [::demo::Note require_folder \
        -name demo-note -package_id [my id]]
}
...
```

Unit 1 / Best Practice

Best practices

- . Always provide a application-specific package manager, it facilitates both content management and request processing (in www/* scripts)
- . Declare your package manager in a library (*-procs.tcl) script, e.g. tcl/note-procs.tcl
- . It is good practice to declare application-specific code in a proper TCL namespace; in our example everything goes into: namespace eval ::demo { ... }

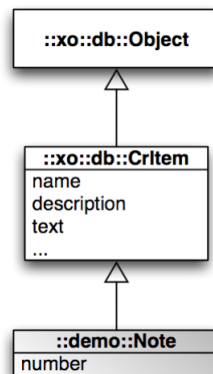
End of Unit 1



Unit 2 / Devise your Application's Data Model

- . Remember the section on XOTcl Core's Interface to the content repository presented earlier.
- . Notes will be realized as special kind of Content Items in the Content Repository
 - . On the one hand, this allows to seamlessly integrate framework features such as comments, notifications, category management, full-text searches etc. with our notes application.
 - . On the other hand, we gain all benefits from the Content Repository (access, revision management, structuring, ...)
- . By using the OO Content Repository interface it is sufficient to define a new type of content item. The SQL data model is created automatically
- . Let's consider our application data model conceptually

Unit 2 / Conceptual Content Model



Unit 2 / Define a Package-specific Content Type

```

# We are still in the Tcl namespace ::demo in file note-procs.tcl
...
# The class ::demo::Note has an additional attribute "number" in
# addition to the common attributes of CrItems. Since Note is a
# subclass of content repository items, the additional attributes
# are versioned, this means that when an entry is modified and
# saved, the old revisions with the old values are still continue to
# exist.

::xo::db::CrClass create Note -superclass ::xo::db::CrItem \
    -pretty_name "Demo Note" -pretty_plural "Demo Notes" \
    -slots {
        ::xo::db::CrAttribute create number \
            -datatype integer -default 0
    }
}
# end of file note-procs.tcl
ns_log Notice "note-procs.tcl loaded"

```

Unit 2 / Checking the Data Model

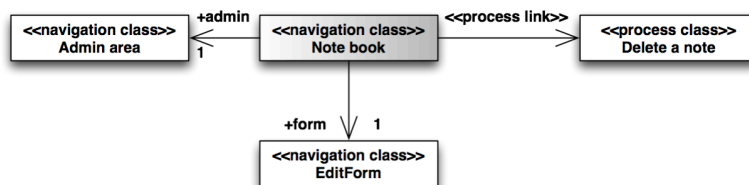
- . After note-procs.tcl is created, it can be loaded into a running OpenACS instance via APM (/acs-admin/apm), choose "Reload files changed" for the note package.
- . This will cause the XOTcl Core to set-up the data model
- . Primarily the following things are created in the database:
 - . A new entry in relation "acs_object_types" called "::demo::Note"
 - . A new attribute table "demo_note" that stores type-specific attribute values, e.g. number.
- . As there is no need for creation SQL scripts, there is also no need for SQL deletion ones. You may use ::demo::Note->drop_object_type manually or in a package removal hook (after-uninstall etc.)

End of Unit 2

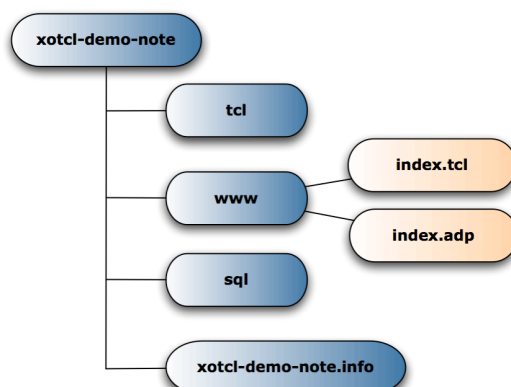


Unit 3 / The Navigational Model

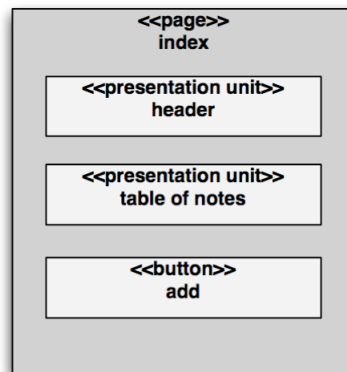
A navigation model as proposed by UML for Web Engineering



Unit 3 / The Index Page as Note Book



Unit 3 / The Presentation Model: Index Page



Unit 3 / Package manager (1)

www/index.tcl: Using your package manager as context resolver

- . Call to a creator method: `::xo::PackageMgr->initialize()`
- . Providing a documentation string
- . Specify expected parameter set

```

# file xotcl-demo-note/www/index.tcl

::demo::Package initialize -ad_doc {

    This is the main page for the package.
    It displays all of the Demo Notes and
    provides links to create, edit and delete Notes.

    @author Gustaf Neumann
    @cvs-id $Id$
} -parameter {
    {-orderby:optional "last_modified,desc"}
}
...
  
```

Unit 3 / Package Manager (2)

::xo::Package provides a simple page-contract interface, similar to `ad_page_contract`

- . Page contracts (see `ad_page_contract`) negotiate both a required and provided feature set between the template (*.adp) and the script (*.tcl) or the request and the script, respectively.
- . The required features include expected request parameters, optional validation, default values etc.
- . The provided feature include properties, i.e. variable that will be accessible within the scope of the affiliated template.

Unit 3 / Package manager (3)

- . Parameter-wise constraints are realised through XOTcl's non-positional arguments
- . Examples:
 - . `-aParameter:optional aDefaultValue`
 - . `-aParameter:required`
 - . `-aParameter:integer`
 - . `-aParameter ""`
 - . `-aParameter:boolean,required false`

Unit 3 / Use xotcl-core Table Widget

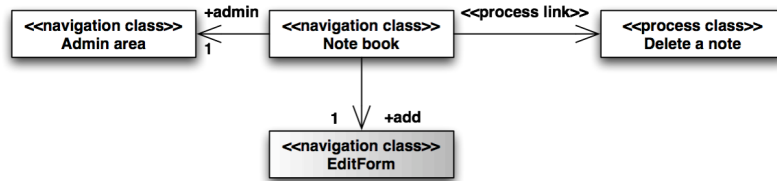
```
# still in file xotcl-demo-note/www/index.tcl
#
# We define a table with an action to add new items
#
TableWidget index -volatile -actions [subst {
    Action new -label Add \
        -url [$package_id package_url]edit \
        -tooltip "Add a new [::demo::Note pretty_name]"
    }} -columns {
    ImageField EditIcon edit -label "" -html {style "padding-right: 2px;"}
    AnchorField name -label "Name" -orderby name
    Field size -label "Size" -orderby size -html {align right}
    Field last_modified -label "Last Modified" -orderby last_modified
    Field mod_user -label "By User" -orderby mod_user
    ImageField DeleteIcon delete -label "" \
        ;#-html {onClick "return(confirm('Confirm delete?'))};"
    }
}
...
```

Unit 3 / Populating the Table Widget

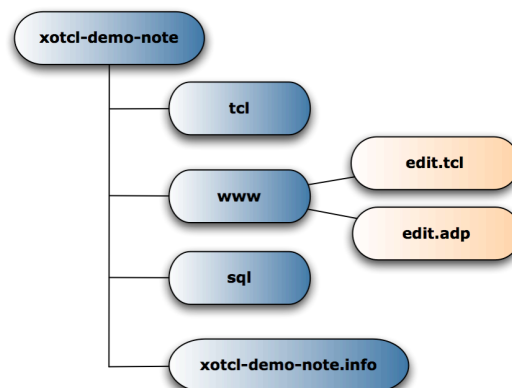
```
# still in file xotcl-demo-note/www/index.tcl
#
# We populate the table widget with notes stored in the db
# We use the ::xo::db::Class->instance_select_query interface
#
db_foreach instance_select [::demo::Note instance_select_query \
    -folder_id [$package_id folder_id] \
    -select_attributes [list content_length creation_user \
        "to_char(last_modified,'YYYY-MM-DD HH24:MI:SS') as last_modified"]] {
    index add \
        -name $name -name.href [export_vars -base edit {item_id}] \
        -last_modified $last_modified \
        -size [expr {$content_length ne "" ? $content_length : 0}] \
        -edit.href [export_vars -base edit {item_id}] \
        -mod_user [::xo::get_user_name $creation_user] \
        -delete.href [export_vars -base delete {item_id}]
}

set html [index asHTML]
```

Unit 3 / User Dialog to Add and Edit Notes



Unit 3 / The Edit&Update User Interface



Unit 3 / Presentation model for the Edit Page

```

<<page>>
  edit
    <<presentation unit>>
      header
    <<presentation unit>>
      edit form
        <<input>>
          name
        <<input>>
          section
        <<input>>
          title
        <<richtext>>
          text
        <<input>>
          description
        <<button>>
          submit
  
```

Unit 3 / Using the Form Builder ::Generic::Form (1)

```

# file xotcl-demo-note/www/admin/edit.tcl
::demo::Package initialize -parameter { {-item_id:integer} }

# This script is called in multiple situations: it is called, when
# an item should be newly created (with default fields), when the
# values for the new item are provided by the user, or when
# an existing item should be displayed or saved. Depending on the
# situation, we might have the item_id of the note. If it exists,
# we can fetch it from the database to use the values from there.

if {[info exists item_id] && [::xo::db::Class exists_in_db -id $item_id]} {
  set item [::demo::Note get_instance_from_db -item_id $item_id]
} else {
  set item [::demo::Note new -package_id $package_id]
  $item set parent_id [$package_id folder_id]
}
...

```

Unit 3 / Using the Form Builder ::Generic::Form (2)

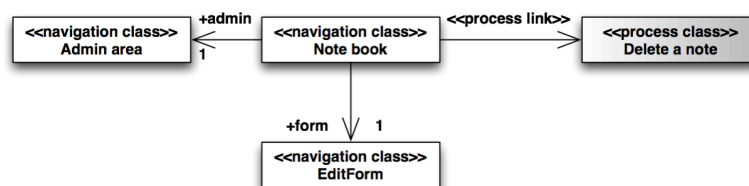
```
# still in file xotcl-demo-note/www/admin/edit.tcl

#
# Provide a form + field specification
#

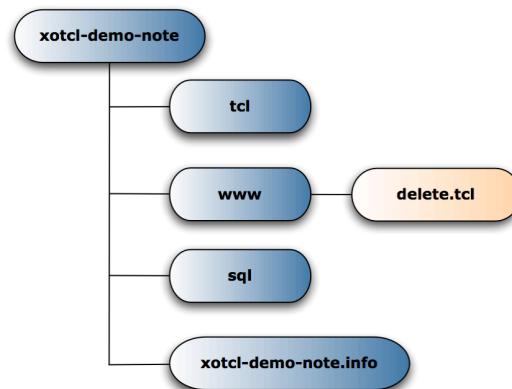
::Generic::Form create form1 -volatile -data $item -fields {
    {item_id:key}
    {name:text {label Name}}
    {number:text {label Number}}
    {text:richtext(richtext),nospell,optional {label Content} {options {editor x
    {description:text(textarea),nospell,optional {label Description}
        {html {cols 60 rows 2}}}
    }
}

form1 generate
# provide values for edit.adp
form1 instvar context formTemplate
set title "My first Form"
```

Unit 3 / Deletion of Notes



Unit 3 / Script for Note Deletion



Unit 3 / Deletion of Notes

```

# file xotcl-demo-note/www/admin/delete.tcl

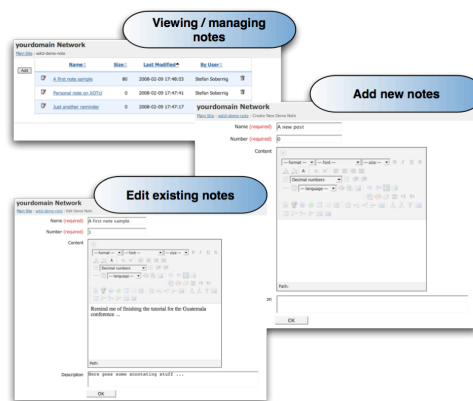
::demo::Package initialize -ad_doc {
    Delete a note
} -parameter {
    {-item_id:integer}
}

#
# We need to verify the current user's privileges
#
permission::require_write_permission -object_id $item_id

#
# Proceed with deletion
#
::xo::db::CrItem delete -item_id $item_id

```

Unit 3 / What did we achieve?



Questions?



References

- . www.xotcl.org
- . Handbook
- . Tutorial
- . Publications Section
- . [UML for Web Engineering](#)
- . <git://alice.wu-wien.ac.at/xotcl-demo-note>