Open Source Health Information Platform

OSHIP

Application Developer's Documentation

Table of Contents

1.Introduction 4
2.Creating the Demo 4
3. The Python Code 5
a.Views - any class that inherits from grok.View: 5
b.Viewlets - any class that inherits from grok.Viewlet: 5
c.ViewletManagers - any class that inherits from grok.ViewletManager: 6
d. Header 7
e. InfoArea 7
f. Patients 7
g. Footer 8
h.Time for a quick recap: 8
4.The Demographic Record 8
1.Adding Patients 8
2.Searching for Patients 9
3.Looking at the code 9
5. The Clinical Record 10
1.Looking at the code 10
6.Component Architecture 11
7.Sources & Vocabularies 11
8.Indexing & Searching 11
9.Changing/encrypting the admin password 11
10.Using the Python interpreter for debugging 12
11.Python Archetypes 12
1.Creating new template code 12
2.Editing Source Templates 12
3.The knowledge Module (km) Tree 13
12.Internationalization/Localization 13

13.Security 13
14.Persistence 13
15.Testing 13
16.Workflows 13
17.Development Tools 16
  1.Python IDE 16
  2.ADL Workbench 16
  3.Interface Design Editor 17
  4.Developer's Page Template Editor 17

OSHIP Application Developer's Guide

1. Introduction

This document will first get you up and running with the demo application and then go deeper into how things work and how to start your own application. It is assumed that you have an OSHIP server running on localhost port 8080 and that you are logged in as admin (the manager login).

We will be using the Blood Pressure Tracker Demo to describe how applications are developed using OSHIP. It is also expected that the developer will be familiar with the Python programming language as well as the Template Attribute Language (TAL) required for data access in HTML pages as used in the Zope Component Architecture and especially in Grok. This is meant to give a verybasic introduction as to how a Grok based application is put together. There are many documents and a great communities to answer more in depth questions about OSHIP/Grok/ZCA, etc. The Resources section of the OSHIP Management page will get you started in learning more about this infrastructure.

Please note that this demo is a work in progress. We do hope tht you will provide feedback in the way of bug reports on Lanuchpad.[1]

2. Creating the Demo

The demo application is created on the Grok applications page. If your OSHIP server is up and running per the installation instructions then you should be able to navigate to http://localhost:8080/applications and login with your admin password. Enter 'bptrack' in the input box labeled: oship.bptrack.bptrack Click the Create button. You can now go to the demo application at http://localhost:8080/bptrack. It will execute a short setup routine (see below) and then redirect to http://localhost:8080/bptrack/bpmain. From this main page you can add a couple of demo patients. Note that the patient name should appear in the listing on the left side of the page. It should be highlighted/underlined as

1 Http://launchpad.net/oship

an HTML link. This link will take you from this patient's specific demographic record to their clinical record. Creating entries in and using the clinical record is described in the section titled The Clinical Record

3. The Python Code

For the demo application, the Python code is contained in bptrack.py. Though it uses a lot of underlying openEHR and Grok/Zope components. It is most important in the beginning that the application developer concentrates on this module. Most of the code is well documented as to what it does. It is however important to have an understanding of how Grok works with regard to Python code interacting with the page templates (HTML+TAL).

a. Views - any class that inherits from grok.View:

These can be thought of as the full page containing the display to the end-user.

b. Viewlets - any class that inherits from grok.Viewlet:

These are the smallest component of an application as far as the end-user is concerned and usually are just snippets of HTML+TAL code that usually define an activity. That activity may be to present output or accept input.

c. ViewletManagers - any class that inherits from grok.ViewletManager:

These are the classes that determine the layout within a View. They are the connectors between Viewlets and Views.

Though View and Viewlet class instances can produce output or perform other activities via a render() method. They are most often connected with a pagetemplate of the same name, in lowercase. For example the class BPMain is a View. It calls bpmain.pt because it has no render method defined. You cannot have both a render method and a page template defined for the same class. Though you MUST have one or the other.

Notice just above BPMain is a class named Index. Index is called by default for any application that does not otherwise have a different pagetemplate defined in it's class declaration. See that bptrack is the application class for this demo and it has no other methods or names assigned. Index does have a render method defined, therefore there is no page template named index.pt. The render method simply adds two grok containers to the application. One for demographic information and one for clinical information. In real life, large applications you could easily have these two components (grok containers) on separate servers or maybe you already have a Master Patient Index (MPI) that you want to connect to for demographics. Therefore you wouldn't even use the demographics container. This is all doable but beyond the scope of this demo.

After creating the containers; Index does a redirect to bpmain, which is the primary View class for this demo. Notice again that there is no render method so Grok looks for the bpmain.pt file in bptrack_templates. Taking a look at the source for bpmain.pt you will see that it basically contains the standard HTML markup expected for a page. The layout is controlled via the bpmain.css file located in the static/css subdirectory. In the body of bpmain.pt, all that is defined is four <div>; sections with TAL attributes pointing to content providers. These content providers are Viewlet Managers. They are:

d. Header

It has one viewlet: pagetitle.pt

e. InfoArea

It has two viewlets:
one for the patient add form (newpatient.pt) and one with just text warning (info.pt) about the fact that this is just a demo. Notice that the corresponding classes in bptrack.py container a call to grok.order() that determines the sequence in the viewlet manager each viewlet will appear. This viewlet manager could have and probably should be divided into two separate managers in real applications.

f. Patients

It has two viewlets:
one for a search box for surnames (patientshead.pt) and one that lists the currently entered patients (PatientsList). Notice that the PatientsList class does not have an associated pagetemplate but a render method. More on this later. In a real world application with tens of thousands of patients this must be more robust and paginated. But there is no point in complicating this basic demo.

g. Footer

It has one viewlet:

footertext.pt which simply contains a copyright notice.

h. Time for a quick recap:

Pages (aka. Views) contain ViewletManagers. These
viewlet managers, defined in <div>s, are laid out using CSS and their
associated id. Viewlets contain snippets of HTML/TAL and are associated with
a specific viewlet manager and have an order specified within that
manager.

Each page template must have an associated View or Viewlet class defined
within the application context (Python code). ViewletManagers are associated
as sections inside a View and are defined as content providers. The id is used to associate it
within the CSS file which is registered in the head of the primary page (I.e. View)..

4. The Demographic Record

The demographic section of this demo is very simple. As stated earlier, it is a grok.Container instance named 'demographic'. It was created when you clicked on the bptrack link from the Applications page by the render method in the Index class.

1. Adding Patients

You add patients by entering a surname a given name and date of birth in the form on the main page of the demo. Note that if you do not enter the DOB per the format in the input box (YYYY/MM/DD) you will receive an error with a traceback ending in:

```
ValueError: time data did not match format:  data=1962/0707  fmt=%Y/%m/%d
```

Notice that the format is not correct because there is a missing slash character between the month and day digits.

When you successfully create a patient, their name will appear as a list item on the left side of the display. It will be underlined because it also contains a link to the clinical' record container where this individual persons clinical records are stored.

2. Searching for Patients

(to be implemented)

3. Looking at the code

The code for adding a patient really starts with defining a patient. That is done in the Interface class for Patient. The interface is implemented by the Patient class. The HTML form for adding a patient instance only passes three pieces of information. But the class also needs to define a patient ID to specifically identify this person in the system and an EHRID as a link to the correct record in the clinical container. Notice that these two Ids are randomly created by the Python UUID module.

The real work happens as the HTML form Submit button calls the AddPatient class. Notice that this is a grok.View with it's own render method. Therefore there is not an associated page template. The patient is instantiated and then added to the demographics container using the patid attribute as a key. Next, an Ehr() object (a grok.Container) is added to the clinical container using the ehrid as a key. We also add the date/time created and the fact that it was created with OSHIP just to demonstrate that it is a container and this is how to add items to it.

The application then redirects back to the main demo page; bpmain.pt where you see the patient listed. If you place your mouse cursor over the name and look at the addrress bar of your browser you will see that if you click on the link you will go to the encounterview and pass three variables, the ehrid, fullname and dob.

These were created in the render method of the PatientsList class.   Most important is that the ehrid points to the correct record in the clinical container. We also pass fulln name (concatenated from given name + surname and dob just so that we can display these last two on the patient's clinical record. The explanation of how the URL was created is documented in the code.

5. The Clinical Record

Access to the patient's clinical record must be initiated from the Patient Listing. Clicking on the name will take you to the patient record. Maybe we could call this an EMR or just a registry in this case. Whatever name you choose to call this section it is when the clinical findings are recorded in OSHIP applications.

1. Looking at the code

The clinical record does not contain any patient identifying information. The only link is from the demographics record via the patient.ehrid attribute. The 'demographics' container was created when you first accessed bptrack from the applications screen. Each actual patient record is an instance of the Ehr() class named with the ehrid attribute. The Ehr() class is a grok.Container.

The link from the patient name calls the encounterview.pt page template. This template is a grok.View containing three ViewletManagers.

The EncounterHeader uses the encounterpagetitle.pt and patientinfo.pt Viewlets.
The EncounterFooter uses the encounterfooter.pt tempate.
The BPForm is where we attach the Viewlets for putting the information in that will be stored as archetyped data. Notice that in the encounterview.pt, the <div> containing BPForm is entirely inside one HTML form. This allows us to create on Submit for everything.   Section of archetyped information is defined as a Viewlet. If you want to restrict what is on the actual form (and the archetype allows it) you simply comment out the Viewlet. For example, the first

Viewlet is about the device used to do the measurement. If you prefer to not have that appear on the form then change:

```
class DeviceInfo(grok.Viewlet):
    grok.context(bptrack)
    grok.viewletmanager(BPForm)
    grok.order(1)
```

To:

```
#class DeviceInfo(grok.Viewlet):
#    grok.context(bptrack)
#    grok.viewletmanager(BPForm)
#    grok.order(1)
```

The only side effect is that when you start the server you will see a warning that you have unused templates in the directory.

6. Component Architecture

[describe the component architecture of Zope3 and utilities and adapters]

7. Sources & Vocabularies

[describe how sources (short lists) and vocabularies work well with archetypes.]

8. Indexing & Searching

[describe the catalog system in Zope3]

9. Changing/encrypting the admin password

To change the password and encryption method execute bin/zpasswd.

After answering the questions you will see an output similar to this:

```
============================================
Principal information for inclusion in ZCML:
  <principal
    id="admin"
    title="OSHIP Admin"
    login="admin"
    password="17e15cddd033e22ae348aeb5660fc2140aec35850c4da997"
    description="OSHIP Developer"
    password_manager="SHA1"
    />
```

Copy the lines with the password and the password_manager **ONLY** and replace those lines in your oship/etc/site.zcml.in file. Then re-run bin/buildout.

10. Using the Python interpreter for debugging

See information about bin/oship-debug

11. Python Archetypes

1. Creating new template code

2. Editing Source Templates

3. The knowledge Module (km) Tree

12. Internationalization/Localization
To be expanded but see the information on i18nextract.

13. Security
[describe permissions, principles and roles]

14. Persistence
[describe the ZODB, SQL connectors and RelStorage]

15. Testing
[describe the testing framework]

16. Workflows
Workflows can be defined using the ZCA's zope.app.workflow module. However a simpler workflow engine has been included with OSHIP. It is named hurry.workflow more information about it is available in the EGG-INFO/PKG-INFO file in the hurry.workflow egg.

One short coming of hurry.workflow is that it does not support multiple workflows for an object. Sebastian Ware has extended hurry.workflow with this support. The module workflow.py in the oship/utils directory contains this extension. Below is Sebatian's email regarding usage.

```
=================================================================
Hi Tim (and anybody else interested in multiple workflow support)!

I have made a small experiment adding multiple workflow support to
hurry.workflow using named utilities and namned adapters. It requires
a "default" unnamed workflow to support versions. Maybe you could look
at this and see if it might be useful.

The general idea is that you have an unnamed default workflow. Then
you can add an arbitrary nr of named workflows. These can be set up
with the same transitions as the default workflow or have other sets
of transitions. I really just wanted to create a very light weight
solution with a minimum of code changes.

I define my workflows like this:

#
# Default workflow (without name)

class Workflow(grok.GlobalUtility, workflow.Workflow):
     grok.provides(IWorkflow)

     def __init__(self):
         super(Workflow, self).__init__(create_workflow())

class WorkflowState(grok.Adapter, workflow.WorkflowState):
     grok.context(IProtonObject)
     grok.provides(interfaces.IWorkflowState)

class WorkflowInfo(grok.Adapter, workflow.WorkflowInfo):
     grok.context(IProtonObject)
     grok.provides(interfaces.IWorkflowInfo)

#
# Named workflow (called "sebastian")

class SpecialWorkflow(grok.GlobalUtility, workflow.Workflow):
     grok.name('sebastian')
     grok.provides(IWorkflow)

     def __init__(self):
         super(SpecialWorkflow, self).__init__(create_workflow())

class SpecialState(grok.Adapter, workflow.WorkflowState):
     grok.context(IProtonObject)
     grok.provides(interfaces.IWorkflowState)
     grok.name("sebastian")
     workflow_name = "sebastian"

class SpecialInfo(grok.Adapter, workflow.WorkflowInfo):
     grok.context(IProtonObject)
     grok.provides(interfaces.IWorkflowInfo)
     grok.name("sebastian")
     workflow_name = "sebastian"

#
# Getting a workflow utility
from zope.component import getUtility
default_wu = getUtility(IWorkflow)
named_wu = getUtility(IWorkflow, "sebastian")

#
# Manipulating an object
from zope.component import getAdapter
default_wi = IWorkflowInfo(item)
named_wi = getAdapter(item, IWorkflowInfo, name="sebastian")
```

```
Attached is the updated hurry.workflow.workflow(.py) file. I also
added the attribute "workflow_name" to interface.py for both
IWorkflowInfo and IWorkflowState.
==========================================================================
```

The OSHIP project needs to gain some experience with this and develop this section of the manual further.

17. Development Tools

It is assumed that you already have a Bazaar client for your platform if you have the OSHIP code and this document. If you obtained this document elsewhere please be aware that OSHIP is hosted at http://launchpad.net/oship and you will need a Bazaar version control client.

These specific or general type tools are highly recommended for building applications on OSHIP. You could of course only use a text editor for all of these functions but it isn't suggested that you do that.

1. Python IDE

If you already use a specific Python IDE then that should be your choice. If you haven't selected one yet, there are several and you should try out a few. I prefer WingIDE Professional even though it isn't open source, the developers are very open source friendly and have a liberal license policy for open source developers.

2. ADL Workbench

The Archetype Definition Language Workbench is an open source product from Ocean Informatics.

This tool is an essential reference when editing your Python source archetypes.
You can get the latest version from
http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/apps/doc/adl_workbench_help.htm
There are links to usage and installation instructions on this page.

You will need to install Eiffel Studio for your platform and then install the ADL Workbench. I use this shell script (on Ubuntu Linux x86_64) in the directory where I installed the workbench to set the environment and start the workbench. You will need to edit it according to your OS and platform.

```
=================================================
#!/bin/bash
export ISE_EIFFEL=/usr/local/Eiffel63
export ISE_PLATFORM=linux-x86-64
export PATH=$PATH:$ISE_EIFFEL/studio/spec/$ISE_PLATFORM/bin
./adl_workbench
=========================================================
```

3. *Interface Design Editor*

The user interface designers can use any HTML editor to create the layout. I recommend Amaya. If the designers use MSWord or OpenOffice; you will find that there is a lot of extra cruft you have to work around to insert your TAL code.

4. *Developer's Page Template Editor*

For the page template programmer I recommend the Bluefish editor. I wish it had built in support for TAL but even at that, it is a very efficient tool for use in this environment.

Alphabetical Index