Personis Documentation

Release 12.9

Bob Kummerfeld and Judy Kay

August 03, 2012

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

- User Models as first class citizens
  - independant of a particular application
  - not just fragments of me locked away in individual systems
  - may be distributed over machines I use or in the cloud (could be a personal cloud)
- Scutability and user control
  - I own my model
  - I control what goes in
  - I control what goes out (releasing parts to applications)
  - I can see my model in meaningful forms
- as new evidence about an aspect is available an application (evidence source) *tells* the user model
- the user model *accretes* the evidence
  - a times stamp is added
  - the source (registered name of the application) is added
  - the evidence type (explicit,...) is added
  - the evidence is appended to a list associated with a component of the model

Resolution:

- When an application needs to know information from the model, it asks the model for the value of the required set of components, it *asks* the model for the required set of components
- At that time
  - A filter selects the evidence allowed to the asker
  - A resolver function interprets the allowed evidence
    - the application may specify the resolver function (from those allowed)
    - Or use default
    - Can be very simple (eg Point Query) or arbitrarily sophisticated (eg use Bayesian model, ontology.)
- Embrace inconsistency, multiple interpretations!

Scrutability:

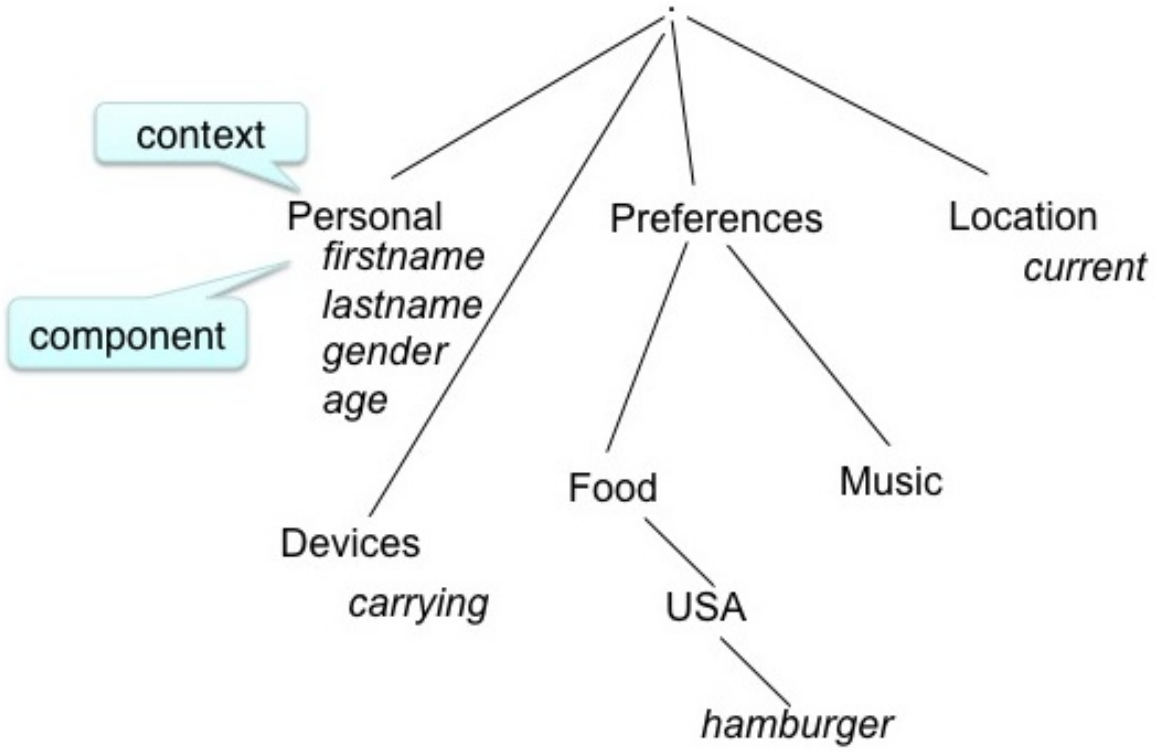
Definition:

> Capable of being understood through study and observation, comprehensible. (www.thefreedictionary.com/scrutable)
>
> Understandable upon close examination. (www.tiscali.co.uk/reference/dictionaries/difficultwords/data/d0011288.html)

- the Personis Framework is designed for scrutability
  - Why did the system adapt that way?
  - Where does the system think I am, and why?
  - Historic queries: what location did the system think I was on May 1st 2001?
  - what music does the system think I like and why?

Model Structure:

- the model is represented as a tree
- we call the branches *contexts* and the leaves *components*



Atomic modelled unit - component:

The components of a model contain the evidence associated with that attribute. Example components:

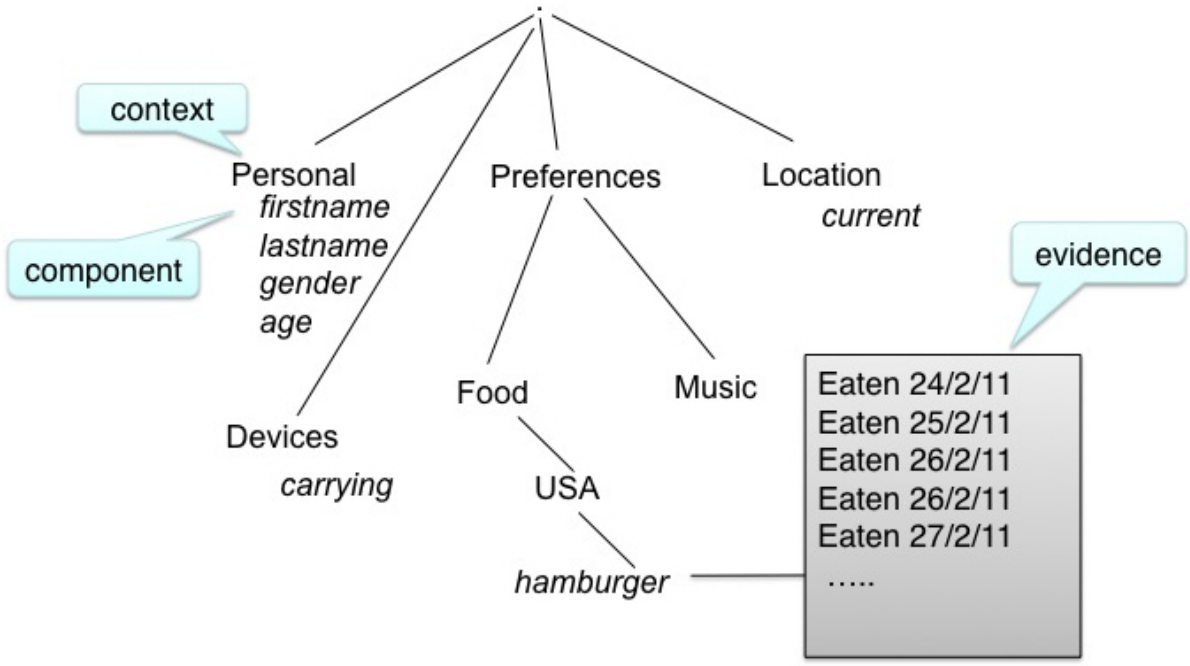
- for classic user model:
  - knowledge
  - beliefs
  - preferences
- for pervasive computing:
  - attributes (eg weight, location, sensor reading)

- qualifiers for knowledge and attributes
- goals (eg I want to be able to do 10 chin-ups)

Operation:

tell

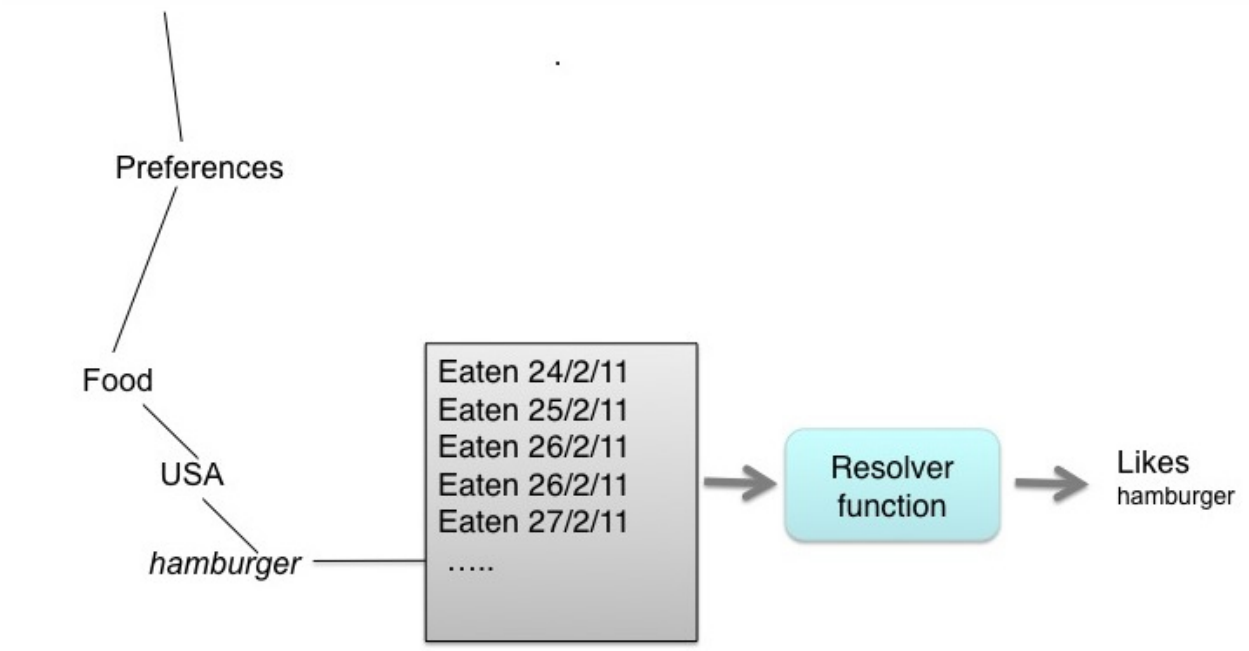
- evidence is accreted by components after the *tell* operation



Operation:

ask

- A component value is retrieved from the nodel using an *ask* operation
  - the evidence is *resolved* by a resolver function to give the value



Accretion/Resolution

- values are only calculated from the evidence when they are needed, rather than whenever a new data point (evidence) is received
- the choice of resolver function allows flexibility in the calculated values. For example, location may be resolved as:
  - room 123, or "at work", or latitude/longitude
- historic queries are possible: where was I on a certain date, how have my music preferences changed.

CHAPTER

TWO

DOCUMENTATION

The documentation for Personis is built using the *sphinx*, a python based documentation generation tool. Sphinx can make the documentation in many forms. To use the tool you need to install sphinx and its dependancies. A generated pdf version is included in the distribution. If you want to make the pdf version you will need to install a full latex package (eg texlive-full).

To make the various versions the command is:

make html
make latexpdf
make epub

The generated versions are placed in the subdirectory "_build".

CHAPTER

THREE

INSTALLATION

These installation instructions assume that you are installing Personis on a linux system with Python 2.6 or 2.7 installed.

The Personis framework makes use of several packages that are not part of the default Python installation. The packages are:

- cherrypy (version 3.x)
- pyparsing
- google oauth2
- httplib2

Personis is known to work with these versions and copies are included in the Personis distribution for your convenience. Personis may work with more recent versions but this has not been tested.

Quick installation instructions:

The distribution of personis is a compressed tar file: Personis-rrr.tgz

First untar the distribution (*rrr* is the release number):

```
$ tar zxf Personis-rrr.tgz
```

This will create a directory Personis-rrr for the code:

```
$ cd Personis-rrr
$ ls
Apps            Personis        README          install.sh
```

To install Personis, use the command:

```
$ ./install.sh
```

The Personis modules are in the Personis/Src directory and can be run directly from that directory or any other directory by setting your PYTHONPATH environment variable. Necessary support packages will be installed in the Personis/lib/python directory and so that directory must also be in the PYTHONPATH.

The following commands will set the path (where *rrr* is the release number):

```
cd Personis-rrr
export PYTHONPATH=`pwd`/Personis/Src:`pwd`/Personis/lib/python
```

You might want to note the value of PYTHONPATH and add a line to your .bashrc to set it for future use.

CHAPTER

FOUR

TESTS

The test scripts are divided into those that test the base Personis module for models stored on the local machine (base), and those that test the server Personis module for models accessed via a network connection (server). In both cases the driver test script gives you the option of nominating a directory to store test models. In the base case the models are accessed directly by the methods in the Personis_base module. In the server case, a Personis server is started to provide access to the models.

To run the tests, change into the Personis directory and:

```
$ bash Tests/base-tests
PYTHONPATH is ...../Personis/Src
model directory? [..../Tests/Models]
```

at this point the test script is asking for a directory to store the test models. The default directory is "Models" in the Tests directory. You can accept this default by pressing return, or type another pathname.

You are now given the option of removing any models previously placed in the test directory. This is useful if you want to rerun the tests from scratch and create the models again. If you press return you will get the default response of "No", typing "Y" will remove the existing models:

```
Remove models in ...../Src/models? [N]
```

If you say Yes (or when you run it for the first time) the models will be recreated. This will produce a lot of output showing all the model parts as they are created.

Next you are given the option of running all the available base tests, running a particular test, or no tests. The base test scripts are stored in the directory Tests/Base and are numbered. If you are testing a particular feature you can type the number, but to start we suggest typing return for all tests. The tests now run, one a time, waiting for you to press return before starting the next. If you want to stop you can press Control-C:

```
Test number? (CR for all, ctrl-C if none)

Running tests...

======================================================================
                          Tests/Base/example01_add.py
======================================================================
add evidence to alice's model
======================================================================
Now check the evidence list for alice's names
======================================================================
Component:  First name
======================================================================
```

```
        ...lots more output...


=================
All Done.
=================
```

There should be only a small number of python error messages and these will have some explanation in the output.

There are a set of similar tests for the server that can be run with the command:

```
$ bash Tests/server-tests

        ...lots of output...

=================
All Done.
=================
```

The models are created in a similar way for the server tests but a server process is started and the client-server protocol used to access them. The server test scripts can be found in Tests/Server.

CHAPTER

FIVE

TUTORIAL INTRODUCTION TO PERSONIS

This tutorial assumes you have installed the framework using the instructions in the Installation section. Using the *umbrowser* command line utility, the tutorial will take you through the construction, navigation and management of a user model.

5.1 umbrowser

The umbrowser.py program is a commandline utility that allows most operations of the Personis system to be carried out interactively.

Umbrowser is found in the clients/browser directory of the personis package and is started with the command ./umbrowser.py:

```
$ ./umbrowser.py
```

The first time you run umbrowser it opens a web browser tab and goes through a login sequence. Usually two screens are presented: one at google that asks you to verify your identity, and one at the personis server asking whether you want to allow umbrowser access to your model (the correct answer is 'yes'). If this is the first time accessing your model, one will be created for you. At the end of this you should end up at a prompt:

```
['']  >
```

The prompt indicates that the browser is waiting for a command. The initial part shows the current context, with the root context as an empty string.

The help command gives information about the available commands:

```
['']  > help
Documented commands (type help <topic>):
========================================
app         delapp        export      login        mkmodel  set       subscribe
attributes  delcomponent  import      ls           model    setgoals  tell
base        delsub        importmdef  mkcomponent  quit     setperm
cd          do            listapps    mkcontext    server   showperm

Undocumented commands:
======================
help
```

Now we can see what is in the model:

```
[''] > ls
Components:
Contexts: [u'Devices', u'Personal', u'Apps', u'Goals']
Views: {}
Subscriptions: {}
```

The "ls" command is a like the Unix ls command: it lists what is in the current context. As you can see, the model is empty - no components, contexts, views or subscriptions. So let's make a new context called "Prefs" in the current (root) context:

```
[''] > mkcontext Prefs
Context description? My preferences
Create new context 'Prefs' in context '['']' with description 'My preferences'
Ok?[N] Y

[''] > ls
Components:
Contexts: [u'Devices', u'Personal', u'Apps', u'Goals', u'Prefs']
Views: {}
Subscriptions: {}
```

Now we will change context to the new "Prefs" context and make a component "food" for our food preferences:

```
[''] > cd Prefs

['', 'Prefs'] > mkcomponent food
Component description? type of food I prefer
Component type:
0 attribute
1 activity
2 knowledge
3 belief
4 preference
5 goal
Index? 4
Value type:
0 string
1 number
2 boolean
3 enum
4 JSON
Index? 0
Creating new component 'food', type 'preference', description 'type of food I prefer', value type 'st
Ok?[N] Y

['', 'Prefs'] > ls
Components:
        food: type of food I prefer
Contexts: []
Views: {}
Subscriptions: {}

['', 'Prefs'] >
```

Now we have a model called "Alice", owned by "alice" that has one context "Prefs" containing one component "food". Now, Alice likes Thai food so we will add some evidence to her food preference component using the "tell" command:

```
Alice ['', 'Prefs'] > tell food
Value? Thai
```

```
Evidence type:
0 explicit
1 implicit
2 exmachina
3 inferred
4 stereotype
Index? [0]
Evidence flag? (CR for none)
Tell value=Thai, type=explicit, flags=[], source=alice, context=['', 'Prefs'], component=food
Ok?[N] Y


Alice ['', 'Prefs'] > ls
Components:
        food: type of food I prefer
Contexts: []
Views: {}
Subscriptions: {}
```

We can now examine the "food" component with the "ls" command:

```
Alice ['', 'Prefs'] > ls food
================================================================
Component:  type of food I prefer
================================================================
showobj:
  Description = type of food I prefer
  component_type = preference
  evidencelist = 1 items
  value_list = []
  value = Thai
  value_type = string
  goals = []
  resolver = None
  Identifier = food
  objectType = Component
--------------------------------
Evidence about it
--------------------------------
showobj:
           comment = None
           evidence_type = explicit
           value = Thai
           objectType = Evidence
           source = alice
           flags = ['']
           time = Thu Apr 28 18:08:55 2011 (1304003335.61)
           owner = alice
           exp_time = 0
           useby = None
--------------------------------
```

Try doing the "tell" operation again with a different food preference and then "ls food" to see the additional evidence that has been accreted.

To quit the model browser, use the *quit* command.

5.2 Logger

On a web browser (your phone will do) go to http://personislog.appspot.com/. Here you will be able to log some activity like eating some fruit. Click on one of the icons to log an activity. Now, let's see what happened to your model.

Start umbrowser, as in the previous section:

```
$ ./umbrowser.py
Welcome James
Personis Model Browser
[''] > ls
Components:
Contexts: [u'Devices', u'Personal', u'Apps', u'Prefs']
Views: {}
Subscriptions: {}
[''] > cd Apps
['', 'Apps'] > ls
Components:
Contexts: [u'Logging']
Views: {}
Subscriptions: {}
['', 'Apps'] > cd Logging
['', 'Apps', 'Logging'] > ls
Components:
        logged_items: All the items logged
Contexts: []
Views: {}
Subscriptions: {}
['', 'Apps', 'Logging'] >
['', 'Apps', 'Logging'] > ls logged_items
```

How did we do this? You can find the source for the logging app, and other personis clients, at https://github.com/jbu/personis/tree/master/clients/ (log-llum is the cherrypy version, aelog is the version that runs on appengine). Look at the method log_me in https://github.com/jbu/personis/blob/master/clients/log-llum/log-llum.py:

```
@cherrypy.expose
def log_me(self, item):
    if cherrypy.session.get('um') == None:
        raise cherrypy.HTTPError(400, 'Log in first.')
    um = cherrypy.session.get('um')
    ev = client.Evidence(source='llum-log', evidence_type="explicit", value=item, time=time.time())
    um.tell(context=['Apps','Logging'], componentid='logged_items', evidence=ev)
    raise cherrypy.HTTPRedirect('/')
```

As you can see, the work is done by two lines. One creates the evidence that something happened, and the next tells the model about it.

CHAPTER

SIX

PERSONIS SERVER

Personis operates as a library that is imported by application programs and stores models in the local file system.

Personis can also be run as a server, providing an interface to models for remote clients. In this case the API is almost the same, the only difference being the modules that is imported and used for the Access call, and the specification of the model to be accessed.

In the case of locally stored models, access requires a *modeldir* argument to specify the location of the stored models, as well as the name of the model (a simple ID). For models accessed remotely via the server, *modeldir* is not used and the model name has the form: name@server[:port].

For example, to access the model for "alice" stored on the server "models.server.com" we would use the statements:

```
import Personis

um = Personis.Access(model="alice@models.server.com", user='myapp', password='pass')
```

6.1 Running a Server

It is very easy to run your own Personis server to provide access to models for remote clients.

A server gets configuration information from the file $HOME/.personis_server.conf. This file specifies the port that the server is to use as well as some miscellaneous configuration options. A suitable personis_server.conf file can be found in the Personis/Src directory. This can be copied into $HOME/.personis_server.conf and the port number changed as desired.

A server can be started for any set of models stored in the same directory using the command:

```
# assuming that PYTHONPATH, MODELDIR and LOGFILE are initialised
Personis.py  --models $MODELDIR --log $LOGFILE &
```

The directory containing the models is specified in $MODELDIR. Log information is written to $LOGFILE. This includes information on all requests, error messages etc.

CHAPTER

SEVEN

EXAMPLE APPLICATIONS

7.1 Museum Guide

7.2 Health Monitoring

7.3 Drill and Practice

CHAPTER

EIGHT

APPLICATION PROGRAM INTERFACE

Files

File	Description
base.py	the core library that accesses models in local directories/files
active.py	adds 'Active User Models' to Personis. This allows components to be 'subscribable' and statements in a simple language to be executed when new component values satisfy a condition.
server.py	a server and set of stubs for the server version of Personis uses active.py to do the work
tests/*	scripts to test the system
mkmodel	utility program to make a set of models from a
modeldef	definition file

Constants

ComponentTypes:

```
"attribute"
"activity"
"knowledge"
"belief"
"preference"
"goal"
```

ValueTypes:

```
"string"
"number"
"boolean"
"enum"
"JSON"
```

EvidenceTypes:

```
"explicit"  # given by the user  (given)
"implicit"  # observed by the machine (observation)
"exmachina" # told (to the user) by the machine (told)
"inferred"  # evidence generated by inference (external or internal)
"stereotype" # evidence added by a stereotype
```

Functions

MkModel(model=None, modeldir=None, user=None, password=None, description=None):

make a model with name “model” in directory modeldir for “user”/”password” with “description”

Classes

```
Component: component object
        Identifier      the identifier of the component
                        unique in the context
        Description     readable description
        component_type  ["attribute", "activity", "knowledge", "belief", "preference", "goal"]
        value_type      ["string", "number","boolean", "enum", "JSON"]
        value_list      a list of strings that are the possible values for type "enum"
        value           the resolved value
        resolver        default resolver for this component
        goals           list of component paths eg [ ['Personal', 'Health', 'weight'], ...]
        evidencelist    list of evidence objects

Evidence: evidence object
        evidence_type   "explicit", # given by the user
                        "implicit", # observed by the machine
                        "exmachina", # told (to the user) by the machine
                        "inferred", # evidence generated by a subscription inference
                        "stereotype"] # evidence added by a stereotype
        source  string indicating source of evidence
        value   any python object
        comment string with extra information about the evidence
        flags   a list of strings eg "goal"
        time    timestamp
        useby   timestamp evidence expires (if required)

Context: context object
        Identifier      the identifier of the component
                        unique in the context
        Description     readable description
        resolver        default resolver for components in this context

View: view object
        Identifier      the identifier of the component
                        unique in the context
        Description     readable description

Access(Resolvers.Access): user model object
        model           model name
        modeldir        model directory
        user            user name
        password        password string
returns a user model access object
```

Access methods:

```
def ask(self,  context=[], view=None, resolver=None, showcontexts=None):
        context is a list giving the path of context identifiers
        view is either:
                an identifier of a view in the context specified
                a list of component identifiers or full path lists
                None indicating that the values of all components in
                        the context be returned
        resolver is a string containing the name of a resolver
                or
```

```
        resolver is a dictionary containing information about resolver(s) to be used and arguments
                the "resolver" key gives the name of a resolver to use, if not present the default re
                the args may include a specified evidence filter
                eg 'evidence_filter' =  "all" returns all evidence,
                                        "last10" returns last 10 evidence items,
                                        "last1" returns most recent evidence item,
                                        None returns no evidence
        showcontexts: if True, a tuple is returned containing
                (list of component objects,
                list of contexts in the current context,
                list of views in the current context,
                list of subscriptions in the current context)
        returns a list of component objects

def tell(self, context=[], componentid=None, evidence=None,   # evidence obj dosubs=True):
        arguments:
                context - a list giving the path to the required context
                componentid - identifier of the component
                evidence - evidence object to add to the component

def export_model(self,  context=[], evidence_filter=None, level=None):
        context is a list giving the path of context identifiers
                this is the root of the um tree to export
        evidence_filter specifies an evidence filter
                (partially implemented: "all" returns all evidence,
                                        "last10" returns last 10 evidence items,
                                        "last1" returns most recent evidence item,
                                        None returns no evidence)
        returns a JSON encoded representation of the um tree

def import_model(self, context=[], partial_model=None):
        arguments:
        context - context to import partial model to
                if None, use root of model
        partial_model - string containing JSON representation of model dictionary
                OR
                a dictionary with elements:
                        contextinfo - Description, Identifier, perms, resolver
                        contexts - sub contexts
                        components
                        views
                        subs

def set_goals(self, context=[], componentid=None, goals=None):
        set the goal list for a component
        requires "tell" permission
        arguments:
                context - a list giving the path to the required context
                componentid - identifier of the component
                goals - list of goal component paths

def mkcomponent(self, context=[], componentobj=None):
        Make a new component in a given context
        arguments:
                context - a list giving the path to the required context
                componentobj - a Component object
        returns:
                None on success
```

```
                a string error message on error

def delcomponent(self, context= [], componentid=None):
        Delete an existing component in a given context
        arguments:
                context - a list giving the path to the required context
                id - the id for a componen
        returns:
                None on success
                a string error message on error

def mkcontext(self, context= [], contextobj=None):
        Make a new context in a given context
        arguments:
                context - a list giving the path to the required context
                contextobj - a Context object
        return True if created ok, False otherwise

def delcontext(self, context=[]):
        Delete an existing context
        arguments:
                context - a list giving the path to the required context
        returns:
                None on success
                a string error message on error

def getcontext(self, context=[], getsize=False):
        get information (Description, size etc) of a context
        arguments:
                context - a list giving the path to the required context
                getsize - if True, return the size in bytes of the context subtree
        returns:
                None on success
                a string error message on error

def registerapp(self, app=None, desc="", password=None):
        registers a password for an app
        app name is a string (needs checking TODO)
        desc is the app description string
        app passwords are stored at the top level .model db
        returns a dictionary containing description and password(access key)

def deleteapp(self, app=None):
        deletes an app

def listapps(self):
        returns an dictionary of apps that are registered
        key is app name, 'description' is app description

def setpermission(self, context=None, componentid=None, app=None, permissions={}):
        sets ask/tell permission for a context (if componentid is None) or
                a component

def setresolver(self, context, componentid, resolver):

def getresolvers(self):

def mkview(self, context= [], viewobj=None):
```

```
        Make a new view in a given context
        arguments:
                context - a list giving the path to the required context
                viewobj - a View object

def delview(self, context=[], viewid=None):
        Delete an existing view within a given context
        arguments:
                context - a list giving the path to the required context
                viewid - view identifier
        returns:
                on success, None
                on failure, a string reporting the problem

def subscribe(context=[], view=None, subscription=None):
        add a subscription to the component specified by the context and view
        arguments:
                context - a list giving the path to the required context
                viewobj - a View object
                subscription - is a dictionary containing owner, password and subscription statement
        returns a token that can be used to delete the subscription

def delete_sub(context=[], componentid=None, subname=None):
        deletes a subscription specified by the token subname in the component specified by the conte
        arguments:
                context - a list giving the path to the required context
                componentid - name of component in the context
                subname - a token return from the subscribe call when the subscription is installed
                                also available using an ask call with showcontexts=True
```

8.1 Examples

Models can be accessed either locally in the filesystem, or via a server.

Local access is via the Personis_base module.

Basic accretion operation - tell some evidence

The following example shows the use of Personis_base to *tell* a piece of evidence containing a name string to a component in the model. The source of the evidence is "contactapp" which will have been given access to the model by the owner.

```
import Personis_base

# access the model in the filesystem
# model name is "alice", model is stored in directory "Models"
um = Personis_base.Access(model="alice", modeldir='Models', user='contactapp', password='secret')

# create a piece of evidence with Alice as value
ev = Personis_base.Evidence(evidence_type="explicit", value="Alice")

# tell this as user alice's first name into component "firstname", context "Personal"
um.tell(context=["Personal"], componentid="firstname", evidence=ev)
```

Basic resolution operation - ask for a value

This example *ask*s for the value of a component using the default resolver that uses the most recent piece of evidence.

```
import Personis_base

um = Personis_base.Access(model="alice", modeldir='Models', user='contactapp', password='secret')

# now ask for the value of the component using the default resolver and the last piece of evidence
reslist = um.ask(context=["Personal"], view=["firstname"], resolver=dict(evidence_filter="last1"))
```

A *view* is just a list of components. The list can be explicit in the ask request or we can give a view a name and store it in the model.

For example:

```
# now ask for the value of two components using a view
reslist = um.ask(context=["Personal"], view=["firstname", "lastname"], resolver=dict(evidence_filter=
```

We can make a view using a view object and the *mkview* method. For example:

```
import Personis_base

um = Personis_base.Access(model="alice", modeldir='Models', user='contactapp', password='secret')

vobj = Personis_base.View(Identifier="fullname", component_list=["firstname", "lastname"])
um.mkview(context=["Personal"], viewobj=vobj)

reslist= um.ask(context=["Personal"], view = 'fullname', resolver={'evidence_filter':"all"})
```

The values are returned by an ask request in a list of component objects, one for eachc component value requested. The component objects have the attributes described in the documentation above but this includes a *value* attribute which is the resolved value for the component. Eg:

```
reslist = um.ask(context=["Personal"], view=["firstname"], resolver=dict(evidence_filter="last1"))
print "Firstname:", reslist[0].value
```

Creating new contexts and components

The *mkcontext* and *mkcomponent* methods, along with the *Component* and *Context* objects, are used to build new elements in the model. Here is an example of creating and then deleting a context:

```
# assume we have accessed the model
print "creating context 'Deltest' in context 'Personal'"
cobj = Personis_base.Context(Identifier="Deltest", Description="testing context deletion")
# now make the new context
um.mkcontext(context=["Personal"], contextobj=cobj)

print "now delete it"
um.delcontext(context=["Personal", "Deltest"]):
```

and here is an example of creating and then deleting a component:

```
cobj = Personis_base.Component(Identifier="age", component_type="attribute", Description="age", goals

um.mkcomponent(context=["Personal"], componentobj=cobj)

# tell some evidence to the new component
ev = Personis_base.Evidence(evidence_type="explicit", value=17)
um.tell(context=["Personal"], componentid='age', evidence=ev)
reslist = um.ask(context=["Personal"], view=['age'], resolver={'evidence_filter':"all"})
print "Age:", reslist[0].value
```

```
# delete the component
resd = um.delcomponent(context=["Personal"], componentid = "age")
```

Navigating the Model

If you want to discover what contexts are present in the model there is a variant on the *ask* method that allows you to get a list of all the *contexts*, *components*, *views* and *subscriptions* that are contained in a given context. Just add the parameter "showcontexts=True" to the *ask* call. Using this call you can start at the root context and walk the tree of contexts discovering the full contents of the model. Eg:

```
print "Show the root context"
info = um.ask(context=[""], showcontexts=True)
```

The return value is a tuple containing (componentlist, contextlist, viewlist, sublist), where each part of the tuple is a list of objects.

Subscriptions: rules for action

A feature of Personis is the ability to add a rule to a component that is examined when ever a *tell* operation is performed on the component. The rule typically examines a resolved value of the component, matching against a pattern. If the pattern is matched an action is initiated. The action can be a *tell* operation to tell some evidence to a component, or a *notify* operation that will construct a URL and fetch it, thus initiating some action at an external web site. Rules can be deleted using the *delete_sub* method.

Note that you need to use Personis_a instead of Personis_base as that is where the subscription methods are found.

For example:

```
import Personis_base
import Personis_a

um = Personis_a.Access(model="alice", modeldir='Models', user='contactapp', password='secret')

# subscription rule that will match firstname against a wildcard pattern (regular expression):
sub = """
<default!./Personal/firstname> ~ '.*' :
         NOTIFY 'http://www.myweb.me/~alice/action.cgi?' 'firstname=' <./Personal/firstname>
"""

# a token identifying the rule is returned
subtoken = um.subscribe(context=["Personal"], view=['firstname'], subscription={'user':'alice', 'pass

ev = Personis_base.Evidence(evidence_type="explicit", value="Alice")
# do a tell. This should cause the action.cgi script to be invoked with the firstame
um.tell(context=["Personal"], componentid='firstname', evidence=ev)

# delete the rule
um.delete_sub(context=["Personal"], componentid='lastname', subname=subtoken)
```

Import and Export of Models

Models can be imported and exported in JSON (JavaScript Object Notation) form using the *export_model* and *import_model* methods:

```
import Personis_base
import Personis_a

um = Personis_a.Access(model="alice", modeldir='Models', user='contactapp', password='secret')

# export a model sub tree to JSON
```

```
# note that all evidence will also be exported.
modeljson = um.export_model(["Personal"], evidence_filter="all")
print modeljson

# import the same model tree but into a different context.
um.import_model(context=["Temp"], partial_model=modeljson)
```

CHAPTER

NINE

MODEL DEFINITION FORMAT

When creating a new model it is possible to build the tree of contexts and components from a template file in "Model Definition Format" or "modeldef". This is useful when installing applications that need their own subtree of contexts and components. The subtree could be built by the application using the *mkcontext* and *mkcomponent* methods but there is also a "bulk" creation script (Src/Utils/mkmodel.py) that reads "modeldef" files and creates the specified contexts, components, views, subscriptions and initial evidence.

Modeldef files consist of a series text lines. Lines that start with # are comments and ignored.

Lines that start with @@ specify a context to be made. The context name is in the form of a pathname starting at the root of the model. For example:

```
@@Personal: description="Personal information"
```

This specifies a context called "Personal" in the root context of the model.

```
@@Personal/Health: description="Health information"
```

This specifies a context called "Health" in the "Personal" context of the model.

A short string description of the context must be included.

After a context (@@) line, that context becomes the *current* context and any additional non-context elements are created in that context until the current context is changed by another @@ line.

Components are specified using lines that begin with two minus signs (–). For example:

```
--firstname: type="attribute", value_type="string", description="First name", [evidence_type="explici
```

this line specifies that a new component called "firstname" is created in the current context. Attributes of the component are specified using name=value elements. Initial evidence for the component is included as a sequence of bracketed sections as shown.

Subscription rules can also be included with a new component using the "rule" attribute. Eg:

```
--email: type="attribute", value_type="string", description="email address",
# create a subscription that will notify when email address changes
        rule="<default!./Personal/email> ~ '*' : NOTIFY 'http://www.somewhere.com/' 'email=' <./Perso
```

lines can be continued by breaking them after a comma.

Views are specified by lines starting with "==" and include a list of component pathnames. Eg:

```
==fullname: firstname, lastname
```

CHAPTER

TEN

CLIENT/SERVER PROTOCOL

This document describes the Personis *access*, *ask* and *tell* calls in terms of the Python method calls.

access:

```
def access(modelname=string, user=string, password=string, version="11.2")
```

The POST URL is then /access and the body is (for example):

```
{"password": "pass", "modelname": "bob", "user": "bob", "version": "11.2"}
```

the return data is:

```
{"result": "ok", "val": true}     -- on success
{"result": "fail", "val": false}     -- on failure
I might change this to include a diagnostic string as the value of val.
```

tell:

```
def tell(modelname=string, user=string, password=string, version="11.2",
        context=list-of-strings, componentid=string, evidence=dict)
```

The URL is /tell

Body example:

```
{"modelname": "bob",
"user": "bob",
"password": "pass",
"version": "11.2",
"evidence": {"comment": null, "evidence_type": "explicit", "value": "Bob",
               "objectType": "Evidence", "source": "demoex2", "flags": [],
               "time": null, "exp_time": 0},
"context": ["Personal"],
"componentid": "firstname"}
```

Note that the evidence dictionary should be extensible, ie not just the fields shown. Keys are strings, values can be strings/None/integer/list-of-strings

The return data is:

```
{"result": "ok", "val": true}     -- on success
{"result": "fail", "val": false}     -- on failure
I might change this to include a diagnostic string as the value of val.[a]
```

ask:

```
def ask(modelname=string, user=string, password=string, version="11.2",
        context=list-of-strings,
        resolve=dict,
        showcontexts=true-or-false,[b]
        view=list-of-(string-or-list-of-string) )
```

The resolver dictionary is extensible, keys and values are strings, *view* is a list of strings or (list of strings)

The URL is /ask

Body example:

```
{"modelname": "bob",
"user": "bob",
"password": "pass",
"version": "11.2",
"context": ["Preferences", "Music", "Jazz", "Artists"],
"showcontexts": null,
"resolver": {"evidence_filter": "all"},
"view": ["Miles_Davis", ["Personal", "firstname"]]}
```

The return data is a dictionary containing a result and val entries like the Access function. The value for "val" is a list of dictionaries, one per component value being returned.

example:

```
{"result": "ok", "val":
 [        {"Description": "First name",
        "component_type": "attribute",
        "evidencelist": null,
        "value_list": null,
        "value": "Bob",
        "value_type": "string",
        "goals": [],
        "resolver": null,
        "Identifier": "firstname",
        "objectType": "Component"},
        {"Description": "Last name",
        "component_type": "attribute",
        "evidencelist": null,
        "value_list": null,
        "value": "Kummerfeld",
        "value_type": "string",
        "goals": [],
        "resolver": null,
        "Identifier": "lastname",
        "objectType": "Component"}
 ]
}
```

The POST requests should be over HTTP for now, HTTPS later. Also, a future stage of development will involve more crypto with signed, encrypted JSON.

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

INDEX

P