AppTools Documentation

Release 3.0.0

Enthought

November 02, 2008

CONTENTS

1 Application Scripting Framework **1**
- 1.1 Framework Concepts . . . 1
- 1.2 Limitations . . . 2
- 1.3 API Overview . . . 2
- 1.4 Implementing Application Scripting . . . 4

2 Permissions Framework - Introduction **7**
- 2.1 Framework Concepts . . . 7
- 2.2 Framework APIs . . . 8
- 2.3 What Do I Need to Reimplement? . . . 8
- 2.4 Deploying Alternative Managers . . . 9
- 2.5 Using the Default Storage Implementations . . . 9

3 Application API **11**
- 3.1 Defining Permissions . . . 11
- 3.2 Applying Permissions . . . 12
- 3.3 Authenticating the User . . . 13
- 3.4 Getting and Setting User Data . . . 14
- 3.5 Integrating Management Actions . . . 14
- 3.6 Writing SecureProxy Adapters . . . 14

4 Default Policy Manager Data API **17**
- 4.1 Overview of IPolicyStorage . . . 17

5 Default User Manager Data API **19**
- 5.1 Overview of IUserStorage . . . 19
- 5.2 Overview of IUserDatabase . . . 19

6 Preferences **21**

7 The Basic Preferences Mechanism **23**
- 7.1 Strings, Glorious Strings . . . 24

8 Preferences and Types **27**

9 Scoped Preferences **29**
- 9.1 Accessing a particular scope . . . 30

10 Further Reading **31**

11 Preferences in Envisage **33**

12 Undo Framework **35**
12.1 Framework Concepts 35
12.2 API Overview 36

CHAPTER

ONE

Application Scripting Framework

The Application Scripting Framework is a component of the Enthought Tool Suite that provides developers with an API that allows traits based objects to be made scriptable. Operations on a scriptable object can be recorded in a script and subsequently replayed.

The framework is completely configurable. Alternate implementations of all major components can be provided if necessary.

1.1 Framework Concepts

The following are the concepts supported by the framework.

- Scriptable Type

  A scriptable type is a sub-type of `HasTraits` that has scriptable methods and scriptable traits. If a scriptable method is called, or a scriptable trait is set, then that action can be recorded in a script and subsequently replayed.

  If the `__init__()` method is scriptable then the creation of an object from the type can be recorded.

  Scriptable types can be explicitly defined or created dynamically from any sub-type of `HasTraits`.

- Scriptable API

  The set of a scriptable type's scriptable methods and traits constitutes the type's scriptable API.

  The API can be defined explicitly using the `scriptable` decorator (for methods) or the `Scriptable` wrapper (for traits).

  For scriptable types that are created dynamically then the API can be defined in terms of one or more types or interfaces or an explicit list of method and trait names. By default, all public methods and traits (ie. those whose name does not begin with an underscore) are part of the API. It is also possible to then explicitly exclude a list of method and trait names.

- Scriptable Object

  A scriptable object is an instance of a scriptable type.

  Scriptable objects can be explicitly created by calling the scriptable type. Alternatively a non-scriptable object can be made scriptable dynamically.

- Script

  A script is a Python script and may be a recording or written from scratch.

  If the creation of scriptable objects can be recorded, then it may be possible for a recording to be run directly by the Python interpreter and independently of the application that made the recording. Otherwise the application must run the script and first create any scriptable objects refered to in the script.

- Binding

  A script runs in a namespace which is, by default, empty. If the scriptable objects refered to in a script are not created by the script (because their type's `__init__()` method isn't scriptable) then they must be created by the application and added to the namespace. Adding an object to the namespace is called binding.

  Scriptable objects whose creation can be recorded will automatically bind themselves when they are created.

  It also possible to bind an object factory rather than the object itself. The factory will be called, and the object created, only if the object is needed by the script when it is run. This is typically used by plugins.

  The name that an object is bound to need bear no relation to the object's name within the application. Names may be dotted names (eg. `aaa.bbb.ccc`) and appropriate objects representing the intermediate parts of such a name will be created automatically.

  An event is fired whenever an object is bound (or when a bound factory is invoked). This allows other objects (eg. an embedded Python shell) to expose scriptable objects in other ways.

- Script Manager

  A script manager is responsible for the recording and subsequent playback of scripts. An application has a single script manager instance which can be explicitly set or created automatically.

1.2 Limitations

In the current implementation scriptable Trait container types (eg. List, Dict) may only contain objects corresponding to fundamental Python types (eg. int, bool, str).

1.3 API Overview

This section gives an overview of the API implemented by the framework. The complete API documentation is available as endo generated HTML.

The example application demonstrates some the features of the framework.

1.3.1 Module Level Objects

`get_script_manager()` The application's script manager is returned. One will be created automatically if needed.

`set_script_manager(script_manager)` The application's script manager will be set to `script_manager` replacing any existing script manager.

`scriptable` This is a decorator used to explicitly mark methods as being scriptable. Any call to a scriptable method is recorded. If a type's `__init__()` method is decorated then the creation of the object will be recorded.

`Scriptable` This is a wrapper for a trait to explicitly mark it as being scriptable. Any change to the value of the trait will be recorded. Simple reads of the trait will not be recorded unless unless the value read is bound to another scriptable trait or passed as an argument to a scriptable method. Passing `has_side_effects=True` when wrapping the trait will ensure that a read will always be recorded.

`create_scriptable_type(script_type, name=None, bind_policy='auto', api=None, includes=None,` This creates a new type based on an existing type but with certain methods and traits marked as being scriptable. Scriptable objects can then be created by calling the type.

`script_type` is the existing, non-scriptable, type. The new type will be a sub-type of it. The `api`, `includes` and `excludes` arguments determine which methods and traits are made scriptable. By default, all public methods and traits (ie. those whose name does not begin with an underscore) are made scriptable.

The `name` and `bind_policy` arguments determine how scriptable objects are bound when they are created. `name` is the name that an object will be bound to. It defaults to the name of `script_type` with the first character forced to lower case. `name` may be a dotted name, eg. `aaa.bb.c`.

`bind_policy` determines what happens if an object is already bound to the name. If it is `auto` then a numerical suffix will be added to the name of the new object. If it is `unique` then an exception will be raised. If it is `rebind` then the object currently bound to the name will be unbound.

`api` is a class or interface (or a list of classes or interfaces) that is used to provide the names of the methods and traits to be made scriptable. The class or interface effectively defines the scripting API.

If `api` is not specified then `includes` is a list of method and trait names that are made scriptable.

If `api` and `includes` are not specified then `excludes` is a list of method and trait names that are *not* made scriptable.

If `script_init` is set then the `__init__()` method is made scriptable irrespective of the `api`, `includes` and `excludes` arguments.

If `script_init` is not set then objects must be explicitly bound and `name` and `bind_policy` are ignored.

make_object_scriptable(obj, api=None, includes=None, excludes=None) This takes an existing unscriptable object and makes it scriptable. It works by calling `create_scriptable_type()` on the the objects existing type and replacing that existing type with the new scriptable type.

See the description of `create_scriptable_type()` for an explanation of the `api`, `includes` and `excludes` arguments.

1.3.2 ScriptManager

The `ScriptManager` class is the default implementation of the `IScriptManager` interface.

bind_event This event is fired whenever an object is bound or unbound. The event's argument implements the `IBindEvent` interface.

recording This trait is set if a script is currently being recorded. It is updated automatically by the script manager.

script This trait contains the text of the script currently being recorded (or the last recorded script if one is not being currently recorded). It is updated automatically by the script manager.

script_updated This event is fired whenever the `script` trait is updated. The event's argument is the script manager.

bind(self, obj, name=None, bind_policy='unique', api=None, includes=None, excludes=None) This method makes an object scriptable and binds it to a name. See the description of `create_scriptable_type()` for an explanation of the `api`, `includes`, `excludes`, `name` and `bind_policy` arguments.

bind_factory(self, factory, name, bind_policy='unique', api=None, includes=None, excludes=N This method binds an object factory to a name. The factory is called to create the object (and make it scriptable) only when the object is needed by a running script. See the description of `create_scriptable_type()` for an explanation of the `name` and `bind_policy` arguments.

run(self, script) This method runs a script in a namespace containing all currently bound objects. `script` is any object that can be used by Python's `exec` statement including a string or a file-like object.

run_file(self, file_name) This method runs a script in a namespace containing all currently bound objects. `file_name` is the name of a file containing the script.

start_recording(self) This method starts the recording of a script.

stop_recording(self) This method stops the recording of the current script.

1.3.3 IBindEvent

The `IBindEvent` interface defines the interface that is implemented by the object passed when the script manager's `bind_event` is fired.

`name` This trait is the name being bound or unbound.

`obj` This trait is the obj being bound to `name` or None if `name` is being unbound.

1.3.4 StartRecordingAction

The `StartRecordingAction` class is a canned PyFace action that starts the recording of changes to scriptable objects to a script.

1.3.5 StopRecordingAction

The `StopRecordingAction` class is a canned PyFace action that ends the recording of changes to scriptable objects to a script.

1.4 Implementing Application Scripting

The key part of supporting application scripting is to design an appropriate scripting API and to ensure than the application itself uses the API so that changes to the data can be recorded. The framework provides many ways to specify the scripting API. Which approach is appropriate in a particular case will depend on when it is a new application, or whether scripting is being added to an existing application, and how complex the application's data model is.

1.4.1 Static Specification

A scripting API is specified statically by the explicit use of the `scriptable` decorator and the `Scriptable` trait wrapper. For example:

```
from enthought.appscripting.api import scriptable, Scriptable
from enthought.traits.api import HasTraits, Int, Str

class DataModel(HasTraits):

    foo = Scriptable(Str)

    bar = Scriptable(Int, has_side_effects=True)

    @scriptable
    def baz(self):
        pass

    def weeble(self)
        pass

# Create the scriptable object.  It's creation won't be recorded because
# __init__() isn't decorated.
obj = DataModel()
```

```
# These will be recorded.
obj.foo = ''
obj.bar = 10
obj.baz()

# This will not be recorded.
obj.weeble()

# This won't be recorded unless 'f' is passed to something that is
# recorded.
f = obj.foo

# This will be recorded because we set 'has_side_effects'.
b = obj.bar
```

1.4.2 Dynamic Specification

A scripting API can also be specified dynamically. The following example produces a scriptable object with the same scriptable API as above (with the exception that `has_side_effects` cannot be specified dynamically):

```
from enthought.appscripting.api import create_scriptable_type
from enthought.traits.api import HasTraits, Int, Str

class DataModel(HasTraits):

    foo = Str

    bar = Int

    def baz(self):
        pass

    def weeble(self)
        pass

# Create a scriptable type based on the above.
ScriptableDataModel = create_scriptable_type(DataModel, excludes=['weeble'])

# Now create scriptable objects from the scriptable type.  Note that each
# object has the same type.
obj1 = ScriptableDataModel()
obj2 = ScriptableDataModel()
```

Instead we could bypass the type and make the objects themselves scriptable as follows:

```
from enthought.appscripting.api import make_object_scriptable
from enthought.traits.api import HasTraits, Int, Str

class DataModel(HasTraits):

    foo = Str

    bar = Int

    def baz(self):
```

```
        pass

    def weeble(self)
        pass

# Create unscriptable objects.
obj1 = DataModel()
obj2 = DataModel()

# Now make the objects scriptable.  Note that each object has a different
# type, each a sub-type of 'DataModel'.
make_object_scriptable(obj1, excludes=['weeble'])
make_object_scriptable(obj2, excludes=['weeble'])
```

With a more sophisticated design we may choose to specify the scriptable API as an interface as follows:

```
from enthought.appscripting.api import make_object_scriptable
from enthought.traits.api import HasTraits, Int, Interface, Str

class DataModel(HasTraits):

    foo = Str

    bar = Int

    def baz(self):
        pass

    def weeble(self)
        pass

class IScriptableDataModel(Interface):

    foo = Str

    bar = Int

    def baz(self):
        pass

# Create an unscriptable object.
obj = DataModel()

# Now make the object scriptable.
make_object_scriptable(obj, api=IScriptableDataModel)
```

1.4.3 Scripting __init__()

Making a type's `__init__()` method has advantages and disadvantages. It means that the creation of scriptable objects will be recorded in a script (along with the necessary `import` statements). This means that the script can be run independently of your application by the standard Python interpreter.

The disadvantage is that, if you have a complex data model, with many interdependencies, then defining a complete and consistent scripting API that allows a script to run independently may prove difficult. In such cases it is better to have the application create and bind the scriptable objects itself.

CHAPTER

TWO

Permissions Framework - Introduction

The Permissions Framework is a component of the Enthought Tool Suite that provides developers with the facility to limit access to parts of an application unless the user is appropriately authorised. In other words it enables and disables different parts of the GUI according to the identity of the user.

The framework includes an API to allow it to be integrated with an organisation's existing security infrastructure, for example to look users up in a corporate LDAP directory.

The framework is completely configurable. Alternate implementations of all major components can be provided if necessary. The default implementations provide a simple local filesystem user database and allows roles to be defined and assigned to users.

The framework **does not** provide any facility for protecting access to data. It is not possible to implement such protection in Python and using the file security provided by a typical operating system.

2.1 Framework Concepts

The following are the concepts supported by the framework.

- Permission

  A permission is the basic tool that a developer uses to specify that access to a part of the application should be restricted. If the current user has the permission then access is granted. A permission may be attached to a PyFace action, to an item of a TraitsUI view, or to a GUI toolkit specific widget. When the user is denied access, the corresponding GUI control is disabled or completely hidden.

- User

  Each application has a current user who is either *authorised* or *unauthorised*. In order to become authorised a user must identify themselves and authenticate that identity.

  An arbitrary piece of data (called a blob) can be associated with an authorised user which (with user manager support) can be stored securely. This might be used, for example, to store sensitive user preferences, or to implement a roaming profile.

- User Manager

  The user manager is responsible for authorising the current user and, therefore, defines how that is done. It also provides information about the user population to the policy manager. It may also, optionally, provide the ability to manage the user population (eg. add or delete users). The user manager must either maintain a persistent record of the user population, or interface with an external user database or directory service.

  The default user manager uses password based authorisation.

  The user manager persists its data in a user database. The default user manager provides an API so that different implementations of the user database can be used (for example to store the data in an RDBMS, or to integrate with an existing directory service). A default user database is provided that pickles the data in a local file.

- Policy Manager

  The policy manager is responsible for assigning permissions to users and for determining the permissions assigned to the current user. To do this it must maintain a persistent record of those assignments.

  The default policy manager supplied with the framework uses roles to make it easier for an administrator to manage the relationships between permissions and users. A role is defined as a named set of permissions, and a user may have one or more roles assigned to them.

  The policy manager persists its data in a policy database. The default policy manager provides an API so that different implementations of the policy database can be used (for example to store the data in an RDBMS). A default policy database is provided that pickles the data in a local file.

- Permissions Manager

  The permissions manager is a singleton object used to get and set the current policy and user managers.

2.2 Framework APIs

The APIs provided by the permissions framework can be split into the following groups.

- Application API

  This part of the API is used by application developers.

- Policy Manager API

  This is the interface that an alternative policy manager must implement. The need to implement an alternative is expected to be very rare and so the API isn't covered further. See the definition of the IPolicyManager interface for the details.

- Default Policy Manager Data API

  This part of the API is used by developers to store the policy's persistent data in a more secure location (eg. on a remote server) than that provided by the default implementation.

- User Manager API

  This is the interface that an alternative user manager must implement. The need to implement an alternative is expected to be very rare and so the API isn't covered further. See the definition of the IUserManager interface for the details.

- Default User Manager Data API

  This part of the API is used by developers to store the user database in a more secure location (eg. on a remote server) than that provided by the default implementation.

The complete API documentation is available as endo generated HTML.

2.3 What Do I Need to Reimplement?

The architecture of the permissions framework comprises several layers, each of which can reimplemented to meet the requirements of a particular environment. Hopefully the following questions and answers will clarify what needs to be reimplemented depending on your environment.

Q: Do you want to use roles to group permissions and assign them to users?

A: If yes then use the supplied PolicyManager, otherwise provide your own IPolicyManager implementation.

Q: Do you want users to be authenticated using a password?

A: If yes then use the supplied UserManager, otherwise provide your own IUserManager implementation.

Q: Does the IUser interface allow you to store all the user specific information you need?

A: If yes then use the supplied UserDatabase, otherwise provide your own IUserDatabase implementation.

Q: Do you want to store your user accounts as pickled data in a local file?

A: If yes then use the supplied default, otherwise provide UserDatabase with your own IUserStorage implementation.

Q: Do you want to store your policy data (ie. roles and role assignments) as pickled data in a local file?

A: If yes then use the supplied default, otherwise provide PolicyManager with your own IPolicyStorage implementation.

2.4 Deploying Alternative Managers

The permissions framework will first try to import the different managers from the `enthought.permissions.external` namespace. The default managers are only used if no alternative was found. Therefore, alternative managers should be deployed as an egg containing that namespace.

Specifically the framework looks for the following classes:

`PolicyManager` from `enthought.permissions.external.policy_manager`

`PolicyStorage` from `enthought.permissions.external.policy_storage`

`UserDatabase` from `enthought.permissions.external.user_database`

`UserManager` from `enthought.permissions.external.user_manager`

`UserStorage` from `enthought.permissions.external.user_storage`

The example server is such a package that provides PolicyStorage and UserStorage implementations that use an XML-RPC based server to provide remote (and consequently more secure) policy and user databases.

2.5 Using the Default Storage Implementations

The default policy and user managers both (again by default) persist their data as pickles in local files called `ets_perms_policydb` and `ets_perms_userdb` respectively. By default these are stored in the application's home directory (ie. that returned by `ETSConfig.application_home`).

Note that this directory is normally in the user's own directory structure whereas it needs to be available to all users of the application.

If the `ETS_PERMS_DATA_DIR` environment variable is set then its value is used instead.

The directory must be writeable by all users of the application.

It should be restated that the default implementations do *not* provide secure access to the permissions and user data. They are useful in a cooperative environment and as working examples.

CHAPTER

THREE

Application API

This section provides an overview of the part of the ETS Permissions Framework API used by application developers. The Permissions Framework example demonstrates the API in use. An application typically uses the API to do the following:

- define permissions
- apply permissions
- user authentication
- getting and setting user data
- integrate management actions.

3.1 Defining Permissions

A permission is the object that determines the user's access to a part of an application. While it is possible to apply the same permission to more than one part of an application, it is generally a bad idea to do so as it makes it difficult to separate them at a later date.

A permission has an id and a human readable description. Permission ids must be unique. By convention a dotted notation is used for ids to give them a structure. Ids should at least be given an application or plugin specific prefix to ensure their uniqueness.

Conventionally all an applications permissions are defined in a single `permissions.py` module. The following is an extract of the example's `permissions.py` module:

```
from enthought.permissions.api import Permission

# Add a new person.
NewPersonPerm = Permission(id='ets.permissions.example.person.new',
        description=u"Add a new person")

# Update a person's age.
UpdatePersonAgePerm = Permission(id='ets.permissions.example.person.age.update',
        description=u"Update a person's age")

# View or update a person's salary.
PersonSalaryPerm = Permission(id='ets.permissions.example.person.salary',
        description=u"View or update a person's salary")
```

3.2 Applying Permissions

Permissions are applied to different parts of an applications GUI. When the user has been granted a permission then the corresponding part of the GUI is displayed normally. When the user is denied a permission then the corresponding part of the GUI is disabled or completely hidden.

Permissions can be applied to TraitsUI view items and to any object which can be wrapped in a `SecureProxy`.

3.2.1 TraitsUI View Items

Items in TraitsUI views have `enabled_when` and `visible_when` traits that are evaluated to determine if the item should be enabled or visible respectively. These are used to apply permissions by storing the relevant permissions in the model so that they are available to the view. The `enabled_when` and `visible_when` traits then simply reference the permission's `granted` trait. The `granted` trait automatically reflects whether or not the user currently has the corresponding permission.

In order for the view to be correctly updated when the user's permissions change (ie. when they become authenticated) the view must use the `SecureHandler` handler. This handler is a simple sub-class of the standard Traits `Handler` class.

The following extract from the example shows a default view of the `Person` object that enables the `age` item when the user has the `UpdatePersonAgePerm` permission and shows the `salary` item when the user has the `PersonSalaryPerm` permission:

```
from enthought.permissions.api import SecureHandler
from enthought.traits.api import HasTraits, Int, Unicode
from enthought.traits.ui.api import Item, View

from permissions import UpdatePersonAgePerm, PersonSalaryPerm

class Person(HasTraits):
    """A simple example of an object model"""

    # Name.
    name = Unicode

    # Age in years.
    age = Int

    # Salary.
    salary = Int

    # Define the default view with permissions attached.
    age_perm = UpdatePersonAgePerm
    salary_perm = PersonSalaryPerm

    traits_view = View(
            Item(name='name'),
            Item(name='age', enabled_when='object.age_perm.granted'),
            Item(name='salary', visible_when='object.salary_perm.granted'),
            handler=SecureHandler)
```

3.2.2 Wrapping in a SecureProxy

Any object can have permissions applied by wrapping it in a `SecureProxy` object. An adapter is used that manages the enabled and visible states of the proxied object according to the current user's permissions. Otherwise the proxy behaves just like the object being proxied.

Adapters are included for the following types of object:

- PyFace actions
- PyFace widgets **FIXME:** TODO
- Qt widgets
- wx widgets

See Writing SecureProxy Adapters for a description of how to write adapters for other types of objects.

The following extract from the example shows the wrapping of a standard PyFace action and the application of the `NewPersonPerm` permission:

```
from enthought.permissions.api import SecureProxy

from permissions import NewPersonPerm

...

    def _new_person_action_default(self):
        """Trait initializer."""

        # Create the action and secure it with the appropriate permission.
        act = Action(name='New Person', on_perform=self._new_person)
        act = SecureProxy(act, permissions=[NewPersonPerm])

        return act
```

A `SecureProxy` also accepts a `show` argument that, when set to `False`, hides the object when it becomes disabled.

3.3 Authenticating the User

The user manager supports the concept of the current user and is responsible for authenticating the user (and subsequently unauthorising the user if required).

The code fragment to authenticate the current user is:

```
from enthought.permissions.api import get_permissions_manager

get_permissions_Manager().user_manager.authenticate_user()
```

Unauthorising the current user is done using the `unauthenticate_user()` method.

As a convenience two PyFace actions, called `LoginAction` and `LogoutAction`, are provided that wrap these two methods.

As a further convenience a PyFace menu manager, called `UserMenuManager`, is provided that contains all the user and management actions (see below) in the permissions framework. This is used by the example.

The user menu, login and logout actions can be imported from `enthought.permissions.action.api`.

3.4 Getting and Setting User Data

The user manager has a `user` trait that is an object that implements the `IUser` interface. It is only valid once the user has been authenticated.

The `IUser` interface has a `blob` trait that holds any binary data (as a Python string). The data will be read when the user is authenticated. The data will be written whenever it is changed.

3.5 Integrating Management Actions

Both policy and user managers can provide actions that provide access to various management functions. Both have a `management_actions` trait that is a list of PyFace actions that invoke appropriate dialogs that allow the user to manage the policy and the user population appropriately.

User managers also have a `user_actions` trait that is a list of PyFace actions that invoke appropriate dialogs that allow the user to manage themselves. For example, the default user manager provides an action that allows a user to change their password.

The default policy manager provides actions that allows roles to be defined in terms of sets of permissions, and allows users to be assigned one or more roles.

The default user manager provides actions that allows users to be added, modified and deleted. A user manager that integrates with an enterprise's secure directory service may not provide any management actions.

All management actions have appropriate permissions attached to them.

3.6 Writing SecureProxy Adapters

`SecureProxy` will automatically handle most of the object types you will want to apply permissions to. However it is possible to implement additional adapters to support other object types. To do this you need to implement a sub-class of `AdapterBase` and register it.

Adapters tend to be one of two styles according to how the object's enabled and visible states are changed. If the states are changed via attributes (typically Traits based objects) then the adapter will cause a proxy to be created for the object. If the states are changed via methods (typically toolkit widgets) then the adapter will probably modify the object itself. We will refer to these two styles as wrapping adapters and patching adapters respectively.

The following gives a brief overview of the `AdapterBase` class:

`proxied` This instance attribute is a reference to the original object.

`register_adapter(adapter, type, type, ...)` This is a class method that is used to register your adapter and one or more object types that it handles.

`adapt()` This is a method that should be reimplemented by patching adapters. (The default implementation will cause a proxy to be created for wrapping adapters.) This is where any patching of the `proxied` attribute is done. The object returned will be returned by `SecureProxy()` and would normally be the patched object - but can be any object.

`setattr(name, value)` This method should be reimplemented by wrapping adapters to intercept the setting of relevant attributes of the `proxied` object. The default implementation should be used as the fallback for irrelevant attributes.

`get_enabled()` This method must be reimplemented to return the current enabled state.

`set_enabled(value)` This method must be reimplemented to set the enabled state to the given value.

update_enabled(value) This method is called by your adapter to set the desired value of the enabled state. The actual state set will depend on the current user's permissions.

get_visible() This method must be reimplemented to return the current visible state.

set_visible(value) This method must be reimplemented to set the visible state to the given value.

update_visible(value) This method is called by your adapter to set the desired value of the visible state. The actual state set will depend on the current user's permissions.

The `AdapterBase` class is defined in adapter_base.py.

The PyFace action adapter is an example of a wrapping adapter.

The PyQt widget adapter is an example of a patching adapter.

CHAPTER

FOUR

Default Policy Manager Data API

This section provides an overview of the part of the ETS Permissions Framework API used by developers who want to store a policy manager's persistent data in a more secure location (eg. a remote server) than that provided by the default implementation.

The API is defined by the default policy manager which uses roles to make it easier to assign permissions to users. If this API isn't sufficiently flexible, or if roles are inappropriate, then an alternative policy manager should be implemented.

The API is fully defined by the IPolicyStorage interface. The default implementation of this interface stores the policy database as a pickle in a local file.

4.1 Overview of IPolicyStorage

The IPolicyStorage interface defines a number of methods that must be implemented to read and write to the policy database. The methods are designed to be implemented using simple SQL statements.

In the event of an error a method must raise the `PolicyStorageError` exception. The string representation of the exception is used as an error message that is displayed to the user.

CHAPTER

FIVE

Default User Manager Data API

This section provides an overview of the part of the ETS Permissions Framework API used by developers who want to store a user database in a more secure location (eg. a remote server) than that provided by the default implementation.

The API is defined by the default user manager which uses password based authorisation. If this API isn't sufficiently flexible, or if another method of authorisation is used (biometrics for example) then an alternative user manager should be implemented.

The API is fully defined by the IUserDatabase interface. This allows user databases to be implemented that extend the IUser interface and store additional user related data. If the user database is being persisted in secure storage (eg. a remote RDBMS) then this could be used to store sensitive data (eg. passwords for external systems) that shouldn't be stored as ordinary preferences.

In most cases there will be no requirement to store additional user related data than that defined by `IUser` so the supplied UserDatabase implementation (which provides all the GUI code required to implement the IUserDatabase interface) can be used. The UserDatabase implementation delegates the access to the user database to an object implementing the IUserStorage interface. The default implementation of this interface stores the user database as a pickle in a local file.

5.1 Overview of IUserStorage

The IUserStorage interface defines a number of methods that must be implemented to read and write to the user database. The methods are designed to be implemented using simple SQL statements.

In the event of an error a method must raise the `UserStorageError` exception. The string representation of the exception is used as an error message that is displayed to the user.

5.2 Overview of IUserDatabase

The IUserDatabase interface defines a set of `Bool` traits, all beginning with `can_`, that describe the capabilities of a particular implementation. For example, the `can_add_user` trait is set by an implementation if it supports the ability to add a new user to the database.

Each of these capability traits has a corresponding method which has the same name except for the `can_` prefix. The method only needs to be implemented if the corresponding traits is `True`. The method, for example `add_user()` is called by the user manager to implement the capability.

The interface has two other methods.

The `bootstrapping()` method is called by the user manager to determine if the database is bootstrapping. Typically this is when the database is empty and no users have yet been defined. The permissions framework treats this

situation as a special case and is able to relax the enforcement of permissions to allow users and permissions to be initially defined.

The `user_factory()` method is called by the user manager to create a new user object, ie. an object that implements the IUser interface. This allows an implementation to extend the IUser interface and store additional user related data in the object if the `blob` trait proves insufficient.

CHAPTER

SIX

Preferences

The preferences package provides a simple API for managing application preferences. The classes in the package are implemented using a layered approach where the lowest layer provides access to the raw preferences mechanism and each layer on top providing more convenient ways to get and set preference values.

CHAPTER

SEVEN

The Basic Preferences Mechanism

Lets start by taking a look at the lowest layer which consists of the IPreferences interface and its default implementation in the Preferences class. This layer implements the basic preferences system which is a hierarchical arrangement of preferences 'nodes' (where each node is simply an object that implements the IPreferences interface). Nodes in the hierarchy can contain preference settings and/or child nodes. This layer also provides a default way to read and write preferences from the filesystem using the excellent ConfigObj package.

This all sounds a bit complicated but, believe me, it isn't! To prove it (hopefully) lets look at an example. Say I have the following preferences in a file 'example.ini':

```
[acme.ui]
bgcolor = blue
width = 50
ratio = 1.0
visible = True

[acme.ui.splash_screen]
image = splash
fgcolor = red
```

I can create a preferences hierarchy from this file by:

```
>>> from enthought.preferences.api import Preferences
>>> preferences = Preferences(filename='example.ini')
>>> preferences.dump()

 Node() {}
    Node(acme) {}
      Node(ui) {'bgcolor': 'blue', 'ratio': '1.0', 'width': '50', 'visible': 'True'}
        Node(splash_screen) {'image': 'splash', 'fgcolor': 'red'}
```

The 'dump' method (useful for debugging etc) simply 'pretty prints' a preferences hierarchy. The dictionary next to each node contains the node's actual preferences. In this case, the root node (the node with no name) is the preferences object that we created. This node now has one child node 'acme', which contains no preferences. The 'acme' node has one child, 'ui', which contains some preferences (e.g. 'bgcolor') and also a child node 'splash_screen' which also contains preferences (e.g. 'image').

To look up a preference we use:

```
>>> preferences.get('acme.ui.bgcolor')
'blue'
```

If no such preferences exists then, by default, None is returned:

```
>>> preferences.get('acme.ui.bogus') is None
True
```

You can also specify an explicit default value:

```
>>> preferences.get('acme.ui.bogus', 'fred')
'fred'
```

To set a preference we use:

```
>>> preferences.set('acme.ui.bgcolor', 'red')
>>> preferences.get('acme.ui.bgcolor')
'red'
```

And to make sure the preferences are saved back to disk:

```
>>> preferences.flush()
```

To add a new preference value we simply set it:

```
>>> preferences.set('acme.ui.fgcolor', 'black')
>>> preferences.get('acme.ui.fgcolor')
'black'
```

Any missing nodes in a call to 'set' are created automatically, hence:

```
>>> preferences.set('acme.ui.button.fgcolor', 'white')
>>> preferences.get('acme.ui.button.fgcolor')
'white'
```

Preferences can also be 'inherited'. e.g. Notice that the 'splash_screen' node does not contain a 'bgcolor' preference, and hence:

```
>>> preferences.get('acme.ui.splash_screen.bgcolor') is None
True
```

But if we allow the 'inheritance' of preference values then:

```
>>> preferences.get('acme.ui.splash_screen.bgcolor', inherit=True)
'red'
```

By using 'inheritance' here the preferences system will try the following preferences:

```
'acme.ui.splash_screen.bgcolor'
'acme.ui.bgcolor'
'acme.bgcolor'
'bgcolor'
```

7.1 Strings, Glorious Strings

At this point it is worth mentioning that preferences are *always* stored and returned as strings. This is because of the limitations of the traditional '.ini' file format i.e. they don't contain any type information! Now before you start

panicking, this doesn't mean that all of your preferences have to be strings! Currently the preferences system allows, strings(!), booleans, ints, longs, floats and complex numbers. When you store a non-string value it gets converted to a string for you, but you *always* get a string back:

```
>>> preferences.get('acme.ui.width')
'50'
>>> preferences.set('acme.ui.width', 100)
>>> preferences.get('acme.ui.width')
'100'

>>> preferences.get('acme.ui.visible')
'True'
>>> preferences.set('acme.ui.visible', False)
>>> preferences.get('acme.ui.visible')
'False'
```

This is obviously not terribly convenient, and so the following section discusses how we associate type information with our preferences to make getting and setting them more natural.

CHAPTER

EIGHT

Preferences and Types

As mentioned previously, we would like to be able to get and set non-string preferences in a more convenient way. This is where the PreferencesHelper class comes in.

Let's take another look at 'example.ini':

```
[acme.ui]
bgcolor = blue
width = 50
ratio = 1.0
visible = True

[acme.ui.splash_screen]
image = splash
fgcolor = red
```

Say, I am interested in the preferences in the 'acme.ui' section. I can use a preferences helper as follows:

```
from enthought.preferences.api import PreferencesHelper

class SplashScreenPreferences(PreferencesHelper):
    """ A preferences helper for the splash screen. """

    PREFERENCES_PATH = 'acme.ui'

    bgcolor = Str
    width   = Int
    ratio   = Float
    visible = Bool

>>> preferences = Preferences(filename='example.ini')
>>> helper = SplashScreenPreferences(preferences=preferences)
>>> helper.bgcolor
'blue'
>>> helper.width
100
>>> helper.ratio
1.0
>>> helper.visible
True
```

And, obviously, I can set the value of the preferences via the helper too:

```
>>> helper.ratio = 0.5
```

And if you want to prove to yourself it really did set the preference:

```
>>> preferences.get('acme.ui.ratio')
'0.5'
```

Using a preferences helper you also get notified via the usual trait mechanism when the preferences are changed (either via the helper or via the preferences node directly:

```
def listener(obj, trait_name, old, new):
    print trait_name, old, new

>>> helper.on_trait_change(listener)
>>> helper.ratio = 0.75
ratio 0.5 0.75
>>> preferences.set('acme.ui.ratio', 0.33)
ratio 0.75 0.33
```

If you always use the same preference node as the root of your preferences you can also set the class attribute 'PreferencesHelper.preferences' to be that node and from then on in, you don't have to pass a preferences collection in each time you create a helper:

```
>>> PreferencesHelper.preferences = Preferences(filename='example.ini')
>>> helper = SplashScreenPreferences()
>>> helper.bgcolor
'blue'
>>> helper.width
100
>>> helper.ratio
1.0
>>> helper.visible
True
```

CHAPTER

NINE

Scoped Preferences

In many applications the idea of preferences scopes is useful. In a scoped system, an actual preference value can be stored in any scope and when a call is made to the 'get' method the scopes are searched in order of precedence.

The default implementation (in the ScopedPreferences class) provides two scopes by default:

1. The application scope

This scope stores itself in the 'ETSConfig.application_home' directory. This scope is generally used when *setting* any user preferences.

1. The default scope

This scope is transient (i.e. it does not store itself anywhere). This scope is generally used to load any predefined default values into the preferences system.

If you are happy with the default arrangement, then using the scoped preferences is just like using the plain old non-scoped version:

```
>>> from enthought.preferences.api import ScopedPreferences
>>> preferences = ScopedPreferences(filename='example.ini')
>>> preferences.load('example.ini')
>>> p.dump()

  Node() {}
    Node(application) {}
      Node(acme) {}
        Node(ui) {'bgcolor': 'blue', 'ratio': '1.0', 'width': '50', 'visible': 'True'}
          Node(splash_screen) {'image': 'splash', 'fgcolor': 'red'}
    Node(default) {}
```

Here you can see that the root node now has a child node representing each scope.

When we are getting and setting preferences using scopes we generally want the following behaviour:

a) When we get a preference we want to look it up in each scope in order. The first scope that contains a value 'wins'.

b) When we set a preference, we want to set it in the first scope. By default this means that when we set a preference it will be set in the application scope. This is exactly what we want as the application scope is the scope that is persistent.

So usually, we just use the scoped preferences as before:

```
>>> preferences.get('acme.ui.bgcolor')
'blue'
>>> preferences.set('acme.ui.bgcolor', 'red')
```

```
>>> preferences.dump()

  Node() {}
    Node(application) {}
      Node(acme) {}
        Node(ui) {'bgcolor': 'red', 'ratio': '1.0', 'width': '50', 'visible': 'True'}
          Node(splash_screen) {'image': 'splash', 'fgcolor': 'red'}
    Node(default) {}
```

And, conveniently, preference helpers work just the same with scoped preferences too:

```
>>> PreferencesHelper.preferences = ScopedPreferences(filename='example.ini')
>>> helper = SplashScreenPreferences()
>>> helper.bgcolor
'blue'
>>> helper.width
100
>>> helper.ratio
1.0
>>> helper.visible
True
```

9.1 Accessing a particular scope

Should you care about getting or setting a preference in a particular scope then you use the following syntax:

```
>>> preferences.set('default/acme.ui.bgcolor', 'red')
>>> preferences.get('default/acme.ui.bgcolor')
'red'
>>> preferences.dump()

  Node() {}
    Node(application) {}
      Node(acme) {}
        Node(ui) {'bgcolor': 'red', 'ratio': '1.0', 'width': '50', 'visible': 'True'}
          Node(splash_screen) {'image': 'splash', 'fgcolor': 'red'}
    Node(default) {}
      Node(acme) {}
        Node(ui) {'bgcolor': 'red'}
```

You can also get hold of a scope via:

```
>>> default = preferences.get_scope('default')
```

And then perform any of the usual operations on it.

CHAPTER

TEN

Further Reading

So that's a quick tour around the basic useage of the preferences API. For more imformation about what is provided take a look at the API documentation.

If you are using Envisage to build your applications then you might also be interested in the Preferences in Envisage section.

CHAPTER

ELEVEN

Preferences in Envisage

This section discusses how an Envisage application uses the preferences mechanism. Envisage tries not to dictate too much, and so this describes the default behaviour, but you are free to override it as desired.

Envisage uses the default implementation of the ScopedPreferences class which is made available via the application's 'preferences' trait:

```
>>> application = Application(id='myapplication')
>>> application.preferences.set('acme.ui.bgcolor', 'yellow')
>>> application.preferences.get('acme.ui.bgcolor')
'yellow'
```

Hence, you use the Envisage preferences just like you would any other scoped preferences.

It also registers itself as the default preferences node used by the PreferencesHelper class. Hence you don't need to provide a preferences node explicitly to your helper:

```
>>> helper = SplashScreePreferences()
>>> helper.bgcolor
'blue'
>>> helper.width
100
>>> helper.ratio
1.0
>>> helper.visible
True
```

The only extra thing that Envisage does for you is to provide an extension point that allows you to contribute any number of '.ini' files that are loaded into the default scope when the application is started.

e.g. To contribute a preference file for my plugin I might use:

```
class MyPlugin(Plugin):
    ...

    @extension_point('enthought.envisage.preferences')
    def get_preferences(self, application):
        return ['pkgfile://mypackage:preferences.ini']
```

CHAPTER

TWELVE

Undo Framework

The Undo Framework is a component of the Enthought Tool Suite that provides developers with an API that implements the standard pattern for do/undo/redo commands.

The framework is completely configurable. Alternate implementations of all major components can be provided if necessary.

12.1 Framework Concepts

The following are the concepts supported by the framework.

- Command

  A command is an application defined operation that can be done (i.e. executed), undone (i.e. reverted) and redone (i.e. repeated).

  A command operates on some data and maintains sufficient state to allow it to revert or repeat a change to the data.

  Commands may be merged so that potentially long sequences of similar commands (e.g. to add a character to some text) can be collapsed into a single command (e.g. to add a word to some text).

- Macro

  A macro is a sequence of commands that is treated as a single command when being undone or redone.

- Command Stack

  A command is done by pushing it onto a command stack. The last command can be undone and redone by calling appropriate command stack methods. It is also possible to move the stack's position to any point and the command stack will ensure that commands are undone or redone as required.

  A command stack maintains a *clean* state which is updated as commands are done and undone. It may be explicitly set, for example when the data being manipulated by the commands is saved to disk.

  Canned PyFace actions are provided as wrappers around command stack methods to implement common menu items.

- Undo Manager

  An undo manager is responsible for one or more command stacks and maintains a reference to the currently active stack. It provides convenience undo and redo methods that operate on the currently active stack.

  An undo manager ensures that each command execution is allocated a unique sequence number, irrespective of which command stack it is pushed to. Using this it is possible to synchronise multiple command stacks and restore them to a particular point in time.

An undo manager will generate an event whenever the clean state of the active stack changes. This can be used to maintain some sort of GUI status indicator to tell the user that their data has been modified since it was last saved.

Typically an application will have one undo manager and one undo stack for each data type that can be edited. However this is not a requirement: how the command stack's in particular are organised and linked (with the user manager's sequence number) can need careful thought so as not to confuse the user - particularly in a plugin based application that may have many editors.

To support this typical usage the PyFace `Workbench` class has an `undo_manager` trait and the PyFace `Editor` class has a `command_stack` trait. Both are lazy loaded so can be completely ignored if they are not used.

12.2 API Overview

This section gives a brief overview of the various classes implemented in the framework. The complete API documentation is available as endo generated HTML.

The example application demonstrates all the major features of the framework.

12.2.1 UndoManager

The `UndoManager` class is the default implementation of the `IUndoManager` interface.

`active_stack` This trait is a reference to the currently active command stack and may be None. Typically it is set when some sort of editor becomes active.

`active_stack_clean` This boolean trait reflects the clean state of the currently active command stack. It is intended to support a "document modified" indicator in the GUI. It is maintained by the undo manager.

`stack_updated` This event is fired when the index of a command stack is changed. A reference to the stack is passed as an argument to the event and may not be the currently active stack.

`undo_name` This Unicode trait is the name of the command that can be undone, and will be empty if there is no such command. It is maintained by the undo manager.

`redo_name` This Unicode trait is the name of the command that can be redone, and will be empty if there is no such command. It is maintained by the undo manager.

`sequence_nr` This integer trait is the sequence number of the next command to be executed. It is incremented immediately before a command's `do()` method is called. A particular sequence number identifies the state of all command stacks handled by the undo manager and allows those stacks to be set to the point they were at at a particular point in time. In other words, the sequence number allows otherwise independent command stacks to be synchronised.

`undo()` This method calls the `undo()` method of the last command on the active command stack.

`redo()` This method calls the `redo()` method of the last undone command on the active command stack.

12.2.2 CommandStack

The `CommandStack` class is the default implementation of the `ICommandStack` interface.

`clean` This boolean traits reflects the clean state of the command stack. Its value changes as commands are executed, undone and redone. It may also be explicitly set to mark the current stack position as being clean (when data is saved to disk for example).

undo_name This Unicode trait is the name of the command that can be undone, and will be empty if there is no such command. It is maintained by the command stack.

redo_name This Unicode trait is the name of the command that can be redone, and will be empty if there is no such command. It is maintained by the command stack.

undo_manager This trait is a reference to the undo manager that manages the command stack.

push(command) This method executes the given command by calling its `do()` method. Any value returned by `do()` is returned by `push()`. If the command couldn't be merged with the previous one then it is saved on the command stack.

undo(sequence_nr=0) This method undoes the last command. If a sequence number is given then all commands are undone up to an including the sequence number.

redo(sequence_nr=0) This method redoes the last command and returns any result. If a sequence number is given then all commands are redone up to an including the sequence number and any result of the last of these is returned.

clear() This method clears the command stack, without undoing or redoing any commands, and leaves the stack in a clean state. It is typically used when all changes to the data have been abandoned.

begin_macro(name) This method begins a macro by creating an empty command with the given name. The commands passed to all subsequent calls to `push()` will be contained in the macro until the next call to `end_macro()`. Macros may be nested. The command stack is disabled (ie. nothing can be undone or redone) while a macro is being created (ie. while there is an outstanding `end_macro()` call).

end_macro() This method ends the current macro.

12.2.3 ICommand

The `ICommand` interface defines the interface that must be implemented by any undoable/redoable command.

data This optional trait is a reference to the data object that the command operates on. It is not used by the framework itself.

name This Unicode trait is the name of the command as it will appear in any GUI element (e.g. in the text of an undo and redo menu entry). It may include `&` to indicate a keyboard shortcut which will be automatically removed whenever it is inappropriate.

__init__(*args) If the command takes arguments then the command must ensure that deep copies should be made if appropriate.

do() This method is called by a command stack to execute the command and to return any result. The command must save any state necessary for the `undo()` and `redo()` methods to work. It is guaranteed that this will only ever be called once and that it will be called before any call to `undo()` or `redo()`.

undo() This method is called by a command stack to undo the command.

redo() This method is called by a command stack to redo the command and to return any result.

merge(other) This method is called by the command stack to try and merge the `other` command with this one. True should be returned if the commands were merged. If the commands are merged then `other` will not be placed on the command stack. A subsequent undo or redo of this modified command must have the same effect as the two original commands.

12.2.4 AbstractCommand

`AbstractCommand` is an abstract base class that implements the `ICommand` interface. It provides a default implementation of the `merge()` method.

12.2.5 CommandAction

The `CommandAction` class is a sub-class of the PyFace `Action` class that is used to wrap commands.

`command` This callable trait must be set to a factory that will return an object that implements `ICommand`. It will be called when the action is invoked and the object created pushed onto the command stack.

`command_stack` This instance trait must be set to the command stack that commands invoked by the action are pushed to.

`data` This optional trait is a reference to the data object that will be passed to the `command` factory when it is called.

12.2.6 UndoAction

The `UndoAction` class is a canned PyFace action that undoes the last command of the active command stack.

12.2.7 RedoAction

The `RedoAction` class is a canned PyFace action that redoes the last command undone of the active command stack.

- *Search Page*