Extending FreeIPA

Alexander Bokovoy <abokovoy@redhat.com>

Table of Contents

Introduction

FreeIPA is an integrated security information management solution. There is a common framework written in Python to command LDAP server provided by a 389-ds project, certificate services of a Dogtag project, and a MIT Kerberos server, as well as configuring various other services typically used to maintain integrity of an enterprise environment, like DNS and time management (NTP). The framework is written in Python, runs at a server side, and provides access via command line tools or web-based user interface.

As core parts of the framework are implemented as pluggable modules, it is possible to extend FreeIPA on multiple levels. This document attempts to present general ideas and ways to make use of most of extensibility points in FreeIPA.

For information management solutions extensibility could mean multiple things. Information objects that are managed could be extended themselves or new objects could be added. New operations on existing objects might become needed or certain aspects of an object should be hidden in a specific environment. All these tasks may require quite different approaches to implement.

Following chapters will cover high-level design of FreeIPA and dive into details of its core framework. Knowledge of Python programming language basics is required. Understanding LDAP concepts is desirable, though it is not required for simple extensions as FreeIPA attempts to provide sufficient mapping of LDAP concepts onto less complex structures and Python objects, lowering a barrier to fine tune FreeIPA for the specific use cases.

High level design

FreeIPA core is written in Python programming language. The data is stored in LDAP database, and client-server paradigm is used for managing it. A FreeIPA server instance runs its own LDAP database, provided by 389-ds project (formerly Fedora Directory Server). A single instance of LDAP database corresponds to the single FreeIPA domain. Access to all information stored in the database is provided via FreeIPA server core which is run as a simple WSGI application which uses XML-RPC and JSON to exchange requests with its own clients.

Multiple replicas of the FreeIPA instance can be created on different servers, they are managed with the help of replication mechanisms of 389-ds directory server.

As LDAP database is used for data storage, LDAP's Access Control Model is used to provide privilege separation and Kerberos tickets are used to pass-through assertion of authenticity. As Kerberos server is using the same LDAP database instance, use of Kerberos tickets allows to perform operations against the database on the server if a client is capable to forward such tickets via communication channels selected for the operation.

When FreeIPA client connects to FreeIPA server, a Kerberos ticket is forwarded to the server and operations against LDAP database are performed under identity authenticated when the ticket was issued. As LDAP database also uses Kerberos to establish identity of a client, Access Control Information attributes can be used to limit what entries could be accessed and what operations could be performed.

The approach allows to delegate operations from a FreeIPA client to the FreeIPA server and in general gives FreeIPA server ability to interact with any Kerberos-aware service on behalf of the client. It also allows to keep FreeIPA client side implementation relatively light-weight: all it needs to do is to be able to forward Kerberos ticket, process XML-RPC or JSON, and present resulting responses to the user.

Besides run-time core, FreeIPA includes few configuration tools. These tools are split between server and client. Server-side tools are used when an instance of FreeIPA server is set up and configured, while client-side tools are used to configure client systems. While the server tools are used to configure LDAP database, put proper schema definitions in use, create Kerberos domain, Certificate Authority and configure all corresponding services, client side is more limited to configure PAM/NSS modules to work against FreeIPA server, and make sure that appropriate information about the client host is recorded in FreeIPA databases.

Core plug-in framework

FreeIPA core defines few fundamentals. These are managed objects, their properties, and methods to apply actions to the objects. Methods, in turn, are commands that are associated with a specific object. Additionally, there are commands that do not have directly associated objects and may perform actions over few of those. Objects are stored using data store represented by a back end, and one of most useful back ends is LDAP store back end.

Altogether, set of `Object`, `Property`, `Method`, `Command`, and `Backend` instances represent application programming interface, API, of FreeIPA core framework.

In Python programming language object oriented support is implemented using a fairly simple concept that allows to modify instances in place, extending or removing their properties and methods. While this concept is highly useful, in security-oriented frameworks ability to lock down and trace origins of changes is also important. FreeIPA core attempts to implement locking down feature by artificially making instances of foundation classes read-only after their initialization has happened. If an attempt to modify object happens after it was locked down, an exception is thrown. There are many classes following this pattern.

For example, `ipalib.frontend.Command` class is derived from `ipalib.frontend.HasParam` class that derives from `ipalib.plugable.Plugin` class which, in turn, is derived from `ipalib.base.ReadOnly` class.

As result, every command has typed parameters and can dynamically be added to the framework. At the same time, one cannot modify the properties of the command accidentally once it is instantiated. This protects from modifications and enforces true nature of the commands: they cannot have state that is carried over across multiple calls to the same command unless the state is changing globally the whole environment around.

Environment also holds information about the context of execution. The *context* is important part of the FreeIPA framework as it also defines which methods of the command instance are called in order to perform action. *Context* in itself is defined by the *environment* which gives means to catch and store certain information about execution. As with commands themselves, once instantiated, environment cannot be changed.

By default, for primary FreeIPA use, there are three major contexts defined: server, client, and installer/updates.

server context — plugins are registered and communicate with clients via XML-RPC and JSON listeners. They validate

	any arguments and options defined and then execute whatever action they supposed to perform
client context	plugins are used to validate any arguments and options they take and then forward the request to the FreeIPA server.
installer context, updates context	plugins specific to installation and update are loaded and registered. This context can be used to extend possible operations during set up of FreeIPA server.

A user may define any context they want. FreeIPA names server context as '`server`'. When using the `ipa` command line tool the context is '`cli`'. Server installation tools, in particular, '`ipa-ldap-updater`', use special '`updates`' context to load specialized plugins useful during update of the installed FreeIPA server.

Because these utilities use the same framework they will do the same validation, set default values, and perform other basic actions in all contexts. This can help to save a round-trip when testing for invalid data. However, for client-server communication, the server is always authoritative and can re-define what the client has sent.

Name space

FreeIPA has one special type of read-only objects: `NameSpace`. `NameSpace` class gives an ordered, immutable mapping object whose values can also be accessed as attributes. A `NameSpace` instance is constructed from iterable providing its members, which are simply arbitrary objects with `name` attribute. This attribute must conform to two following rules:

- Its value must be unique among the members of the name space
- Its value must pass the `check_name()` function `ipalib.base` module.

`check_name()` function encodes a simple rule of a lower-case Python identifier that neither starts nor ends with an underscore. Actual regular expression that codifies this rule is `NAME_REGEX` within `ipalib.constants` module.

Once name space is created, it locks itself down and becomes read-only. It means that while original objects accessed through the name space might change, the references to them via name space will stay intact. They cannot be removed or changed to point to other objects.

The name spaces are used widely in FreeIPA core framework. As mentioned earlier, API includes set of objects, commands, and methods. Objects include properties that are defined before lock-down. At object's lock-down parameters are placed into a name space and that locks them down so that no parameter specification can change. Command's parameters and options also locked down and cannot change once command instance is instantiated.

Parameters

`Param` class is used to define attributes, arguments, or options throughout FreeIPA core framework. The `Param` base class is not used directly but rather sub-classed to define properties like passwords or specific data types like `Str` or `Int`.

Instances of classes inherited from `Param` base class give uniform access to the properties required to command line interface, Web UI, and internally to FreeIPA code. Following properties are most important:

name	name of the parameter used internally to address the parameter in Python code. The *name* could include special characters to designate a `Param` spec.
cli_name	optional name of the parameter to use in command line interface. FreeIPA's CLI sets a mechanism to automatically translate from a command line option name to a parameter's *name* if *cli_name* is specified.
label	A short phrase describing the parameter. It is used on the CLI when interactively prompting for the values, and as a label for the form inputs in the Web UI. The *label* should start with an initial capital letter.

doc A long description of the parameter. It is used by the CLI when displaying the help information for a command, and as an extra instruction for the form input on the Web UI. By default the *doc* is the same as the *label* but can be overridden when a `Param` instance is created. As with *label*, *doc* should start with an initial capital letter and additionally should not end with any punctuation.

required If set to `True`, means this parameter is required to supply. All parameters are required by default and that means that *required* property should only be specified when parameter **is not required**.

multivalue if set to `True`, means this parameter can accept a Python's tuple of values. By default all parameters are **single-valued**.

When parameter *name* has any of `?`, `*`, or `+` characters, it is treated as parameter spec and is used to specify whether parameter is required, and should it be multivalued. Following syntax is used:

Spec	Name	Required	Multivalue
'var'	'var'	True	False
'var?'	'var'	False	False
'var*'	'var'	False	True
'var+'	'var'	True	True

Access to the value stored by the `Param` class is given through a callable interface:

```
age = Int('age', label='Age', default=100)
print age(10)
```

Following parameter classes are defined and used throughout FreeIPA framework:

Bool boolean parameters that are stored in Python's `bool` type, therefore, they return either `True` or `False` value. However, they accept `1`, `True` (Python boolean), or Unicode strings '`1`', '`true`' and '`TRUE`' as truth value, and `0`, `False` (Python boolean), or Unicode strings '`0`', '`false`', and '`FALSE`' as false.

Flag boolean parameters which always have default value. Property *default* can be used to set the value. Defaults to `False`:

```
verbose = Flag('verbose', default=True)
```

Int integer parameters that are stored in Python's int type. Two additional properties can be specified when constructing `Int` parameter:

- *minvalue* minimal value that this parameter accepts, defaults to `MININT`
- *maxvalue* maximum value this parameter can accept, defaults to `MAXINT`

Float floating point parameters that are stored in Python's float type. `Float` has the same two additional properties as `Int`. Unlike `Int`, there are no default values for the minimal and maximum boundaries.

Bytes a parameter to represent binary data.

Str parameter representing a Unicode text. Both *Bytes* and *Str* parameters accept following additional properties:

- *minlength* minimal length of the parameter
- *maxlength* maximum length of the parameter
- *length* length of the parameters

pattern — regular expression applied to the parameter's value to check its validness

pattern_errmsg — an error message to show when regular expression check fails

IA5Str — string parameter as defined by RFC 4517. It means all characters of the string must be ASCII characters (7-bit).

Password — parameter to store passwords in Python `unicode` type. *Password* has one additional property:

confirm — boolean specifying whether password should be confirmed when entered. The confirmation is enabled by default.

Enum — parameter can have one of predefined values that are specified with *values* property which is a Python's `tuple`.

For most common case of enumerable strings there are two parameters:

BytesEnum — parameter value should be one of predefined `unicode` strings

StrEnum — equivalent to *BytesEnum*. Originally *BytesEnum* was stored in Python's `str` class instances but to be aligned with Python 3.0 changes both classes moved to store as `unicode`.

When more than one value should be accepted, there is *List* parameter that allows to provide list of strings separated by a separator, default to ','. Also, the *List* parameter skips spaces before the next item in the list unless property *skipspace* is set to False:

```
names = List('names', separator=',', skipspace=True)
names_list = names(u'John Doe, John Lee, Brad Moe')
# names_list is (u'John Doe', u'John Lee', u'Brad Moe')
names = List('names', separator=',', skipspace=False)
names_list = names(u'John Doe, John Lee, Brad Moe')
# names_list is (u'John Doe', u' John Lee', u' Brad Moe')
```

Objects

The data manipulated by FreeIPA is represented by an Object class instances. Instance of an Object class is a collection of properties, accepted parameters, action methods, and a reference to where this object's data is preserved. Each object also has a reference to a property that represents a primary key for retrieving the object.

In addition to properties and parameters, Object class instances hold their labels to use in user interfaces. In practice, there are few differences in how labels are presented depending on whether it is command line interface or a Web UI, but they can be ignored at this point.

To be useful, all Object sub-classes need to override `takes_param` property. This is where most of flexibility of FreeIPA comes from.

takes_param attribute

Properties of every object derived from Object class can be specified manually but FreeIPA gives a handy mechanism to perform descriptive specification. Each `Object` class has `Object.takes_param` attribute which defines a specification of all parameters this object type is accepting.

Next example shows how to create new object type. We create an aquarium tank by defining its dimensions and specifying which fish is living there.

```
1:  from ipalib import api, Object
2:  class tank(Object):
3:      takes_params = (
```

```
 4:          StrEnum('species*', label=u'Species', doc=u'Fish species',
 5:                  values=(u'Angelfish', u'Betta', u'Cichlid', u'Firemouth')),
 6:          Float('height', label=u'Height', doc=u'height in mm', default=400.0),
 7:          Float('width', label=u'Width', doc=u'width in mm', default=400.0),
 8:          Float('depth', label=u'Depth', doc=u'Depth in mm', default=300.0)
 9:      )
10:
11:  api.register(tank)
12:  api.finalize()
13:  print list(api.Object.tank.params)
14:  # ['species', 'height', 'width', 'depth']
```

First we define new class, `tank`, that takes four parameters. On line 11 we register the class in FreeIPA's API instance, api. This creates `tank` object in `api.Object` name space. Many objects can be added into the API up until `api.finalize()` is called as we do on line 12.

When `api.finalize()` is called, all name spaces are locked down and all registered Python objects in those name spaces are also finalized which in turn locks their structure down as well.

As result, once we have finalized our API instance, every registered Object can be accessed through `api.Object.<name>`. Our aquarium tank object now has defined `params` attribute which is a name space holding all `Param` instances. Thus we can introspect and see which parameters this object has.

At this point we can't do anything reasonable with our aquarium tank yet because we haven't defined methods to handle it. In addition, our object isn't very useful as it does not know how to store the information about aquarium's dimensions and species living in it.

Object methods

Methods perform actions on the associated objects. The association of methods and objects is done through naming convention rather than using programming language features. FreeIPA expects methods operating on an object `<name>` to be named `<name>_<action>`:

```
class tank_create(Method):
    def execute(self, **options):
        # create new aquarium tank

api.register(tank_create)

class tank_populate(Method):
    def execute(self, **options):
        # populate the aquarium tank with fish

api.register(tank_populate)
```

As can be seen, each method is a separate Python class. This approach allows to maintain complexity of methods isolated from each other and from the complexity of the objects and their storage which is probably most important aspect due to LDAP complexity overall.

The linking between objects and their methods goes further. All parameters defined for an object, may be used as arguments of the methods without explicit declaration. This means `api.Method.tank_populate` will accept `species` argument.

Methods with storage back ends

In order to store the information, `Object` class instances require a back end. FreeIPA defines several back ends but the ones that could store data are derived of `ipalib.CrudBackend`. CRUD, or *Create*, *Retrieve*, *Update*, and *Delete*, are basic operations that could be performed with corresponding objects.

`ipalib.crud.CrudBackend` is an abstract class, it only defines functions that should be overridden in classes that actually implement the back end operations.

As back end is not used directly, FreeIPA defines methods that could use back end and operate on object's defined by certain criteria. Each method is defined as a separate Python class. As CRUD acronym suggests, there are four base operations: `ipalib.crud.Create`, `ipalib.crud.Retrieve`, `ipalib.crud.Update`, `ipalib.crud.Delete`. In addition, method `ipalib.crud.Search` allows to retrieve all entries that match a given search criteria.

When objects are defined and the back end is known, methods can be used to manipulate information stored by the back end. Most of useful operations combine some of CRUD base operations to perform their tasks.

In order to support flexible way to extend methods, FreeIPA gives special treatment for the LDAP back end. Methods using LDAP back end hide complexity of handling LDAP queries and allow to register user-provided functions that are called before or after method. This mechanism is defined by ipalib.plugins.baseldap.CallbackInterface and used by LDAP-aware CRUD classes, `LDAPCreate`, `LDAPRetrieve`, `LDAPUpdate`, `LDAPDelete`, and an analogue to `ipalib.crud.Search`, `LDAPSearch`. There are also classes that define methods to operate on reverse relationships between objects in LDAP to allow addition or removal of membership information both in forward and reverse directions: `LDAPAddMember`, `LDAPModMember`, `LDAPRemoveMember`, `LDAPAddReverseMember`, `LDAPModReverseMember`, `LDAPRemoveReverseMember`.

Most of CRUD classes are based on a `LDAPQuery` class which generalizes concept of querying a record addressed with a primary key and supports JSON marshalling of the queried attributes and their values.

Base LDAP operation classes implement everything needed to create typical methods to work with self-contained objects stored in LDAP.

LDAPObject class

A large class of objects is LDAPObject. LDAPObject instances represent entries stored in FreeIPA LDAP database instance. They are referenced by their distinguished name, DN, and able to represent complex relationships between entries in LDAP like direct and indirect membership.

Any class derived from LDAPObject needs to re-define few properties so that base class can properly function for the specific object that is defined by the class. Below are commonly redefined properties:

container_dn	DN of the container for this object entries in LDAP. This one usually comes from the environment associated with the API and by default is populated from the `DEFAULT_CONFIG` of `ipalibs.constants`. For example, all accounts are stored under `cn=accounts`, with users are under `cn=users,cn=accounts` and groups are under `cn=groups,cn=accounts`. In case of a new object added, it is reasonable to select its container coordinated to default configuration.
object_class	list of LDAP object classes associated with the object
search_attributes	list of attributes that will be used for search
default_attributes	list of attributes that are always returned by searches
uuid_attribute	an attribute that defines uniqueness of the entry
attribute_members	a dict defining relations between other objects and this one. Key is the name of attribute and value is a list of objects this attribute may refer to. For example, `host` object defines that `memberof` attribute of a host may refer to a `hostgroup`, `netgroup`, `role`, `hbacrule`, or `sudorule` object. In other words, it means that `host` could be a member of any of those objects.

reverse_members — a dict defining reverse relations between this object and other objects. Key is the name of attribute and value is the name of an object that refers to this object with the attribute. For example, `role` object defines that `member` attribute of a `privilege` refers to a `role` object.

password_attributes — list of pairs defining an attribute in LDAP and a property of a Python dictionary representing the LDAP object attributes that will be set accordingly if such attribute exists in the LDAP entry. As passwords have restricted access, often one needs only to know that there is a password set on the entry to perform additional processing.

relationships — a dict defining existing relationship criteria associated with the object. These are used in Web UI to allow filtering of objects by the criteria. The value is defined as a tuple of an UI label and two prefixes: inclusive and exclusive that are prepended to the attribute parameter when options are generated by the framework. LDAPObject defines few default criteria: *member*, *memberof*, *memberindirect*, *memberofindirect*, and objects can redefine or append more. Due to regularity of the design of LDAP objects, default criteria already makes it possible to apply searches almost uniformly: one can ask for membership of a user in a group, as well as for a membership of a role in a privilege without explicitly defining those relationships.

These properties define how translation would go from Python side to and from an LDAP backend.

As an example, let's see how role is defined. This is fully functioning plugin that provides operations on roles:

```
 1:  from ipalib.plugins.baseldap import *
 2:  from ipalib import api, Str, _, ngettext
 3:  from ipalib import Command
 4:  from ipalib.plugins import privilege
 5:
 6:  class role(LDAPObject):
 7:      """
 8:      Role object.
 9:      """
10:      container_dn = api.env.container_rolegroup
11:      object_name = _('role')
12:      object_name_plural = _('roles')
13:      object_class = ['groupofnames', 'nestedgroup']
14:      default_attributes = ['cn', 'description', 'member', 'memberof',
15:          'memberindirect', 'memberofindirect',
16:      ]
17:      attribute_members = {
18:          'member': ['user', 'group', 'host', 'hostgroup'],
19:          'memberof': ['privilege'],
20:      }
21:      reverse_members = {
22:          'member': ['privilege'],
23:      }
24:      rdnattr='cn'
25:
26:      label = _('Roles')
27:      label_singular = _('Role')
28:
29:      takes_params = (
```

```
30:          Str('cn',
31:              cli_name='name',
32:              label=_('Role name'),
33:              primary_key=True,
34:          ),
35:          Str('description',
36:              cli_name='desc',
37:              label=_('Description'),
38:              doc=_('A description of this role-group'),
39:          ),
40:      )
41:
42:  api.register(role)
43:
44:
45:  class role_add(LDAPCreate):
46:      __doc__ = _('Add a new role.')
47:
48:      msg_summary = _('Added role "%(value)s"')
49:
50:  api.register(role_add)
51:
52:
53:  class role_del(LDAPDelete):
54:      __doc__ = _('Delete a role.')
55:
56:      msg_summary = _('Deleted role "%(value)s"')
57:
58:  api.register(role_del)
59:
60:
61:  class role_mod(LDAPUpdate):
62:      __doc__ = _('Modify a role.')
63:
64:      msg_summary = _('Modified role "%(value)s"')
65:
66:  api.register(role_mod)
67:
68:
69:  class role_find(LDAPSearch):
70:      __doc__ = _('Search for roles.')
71:
72:      msg_summary = ngettext(
73:          '%(count)d role matched', '%(count)d roles matched', 0
74:      )
75:
76:  api.register(role_find)
77:
78:
79:  class role_show(LDAPRetrieve):
80:      __doc__ = _('Display information about a role.')
81:
82:  api.register(role_show)
83:
84:
85:  class role_add_member(LDAPAddMember):
86:      __doc__ = _('Add members to a role.')
87:
```

```
 88: api.register(role_add_member)
 89:
 90:
 91: class role_remove_member(LDAPRemoveMember):
 92:     __doc__ = _('Remove members from a role.')
 93:
 94: api.register(role_remove_member)
 95:
 96:
 97: class role_add_privilege(LDAPAddReverseMember):
 98:     __doc__ = _('Add privileges to a role.')
 99:
100:     show_command = 'role_show'
101:     member_command = 'privilege_add_member'
102:     reverse_attr = 'privilege'
103:     member_attr = 'role'
104:
105:     has_output = (
106:         output.Entry('result'),
107:         output.Output('failed',
108:             type=dict,
109:             doc=_('Members that could not be added'),
110:         ),
111:         output.Output('completed',
112:             type=int,
113:             doc=_('Number of privileges added'),
114:         ),
115:     )
116:
117: api.register(role_add_privilege)
118:
119:
120: class role_remove_privilege(LDAPRemoveReverseMember):
121:     __doc__ = _('Remove privileges from a role.')
122:
123:     show_command = 'role_show'
124:     member_command = 'privilege_remove_member'
125:     reverse_attr = 'privilege'
126:     member_attr = 'role'
127:
128:     has_output = (
129:         output.Entry('result'),
130:         output.Output('failed',
131:             type=dict,
132:             doc=_('Members that could not be added'),
133:         ),
134:         output.Output('completed',
135:             type=int,
136:             doc=_('Number of privileges removed'),
137:         ),
138:     )
139:
140: api.register(role_remove_privilege)
```

Extending existing object

As said earlier, until API instance is finalized, objects, methods, and commands can be added, removed, or modified freely. This allows to extend existing objects. Before API is finalized, we cannot address objects through the unified interface as `api.Object.foo`, but for almost all cases an object named `foo` is defined in a plugin `ipalib.plugins.foo`.

1. Add new parameter:

```
1:  from ipalib.plugins.user import user
2:  from ipalib import Str, _
3:  user.takes_params += (
4:         Str('foo',
5:              cli_name='foo',
6:              label=_('Foo'),
7:         ),
8:      )
```

2. Re-define User object label to use organisation-specific terminology in Web UI:

```
1:  from ipalib.plugins.user import user
2:  from ipalib import text
3:
4:  _ = text.GettextFactory(domain='extend-ipa')
5:  user.label = _('Staff')
6:  user.label_singular = _('Engineer')
```

   Note that we re-defined locally _ method to use different `GettextFactory`. As GettextFactory is supporting a single translation domain, all new translation terms need to be placed in a separate translation domain and referred accordingly. Python rules for scoping will keep this symbol as `<package>._` and as nobody imports it explicitly, it will not interfere with the framework's provided `text._`.

3. Assume `/dev/null` as default shell for all new users:

```
 1:  from ipalib.plugins.user import user_add
 2:
 3:  def override_default_shell_cb(self, ldap, dn.
 4:                                entry_attrs, attrs_list,
 5:                                *keys, **options):
 6:      if 'loginshell' in entry_attrs:
 7:          default_shell = [self.api.Object.user.params['loginshell'].default]
 8:          if entry_attrs['loginshell'] == default_shell:
 9:              entry_attrs['loginshell'] = [u'/dev/null']
10:
11:  user_add.register_pre_callback(override_default_shell_cb)
```

The last example exploits a powerful feature available for every method of LDAPObject: registered callbacks.

Extending existing method

For objects stored in LDAP database instance all methods support adding callbacks. A *callback* is a user-provided function that is called at certain point of execution of a method.

There are four types of callbacks:

PRE callback	called before executing the method's action. Allows to modify passed arguments, do additional validation or data transformation and specific access control beyond what is provided by the framework.
POST callback	called after executing the method's action. Allows to analyze results of the action and perform additional actions or modify output.
EXC callback	called in case execution of the method's action caused an execution error. These callbacks provide means to recover from an erroneous execution.
INTERACTIVE callback	called at a client context to allow a command to decide if additional parameters should be requested from an user. This mechanism especially useful to simplify complex interaction when there are several levels of possible scenarios depending on what was provided at a client side.

All callback types are available to any class derived from `CallbackInterface` class. These include all LDAP-based CRUD methods.

Callback registration methods accept a reference to callable and optionally ordering argument `first` (`False` by default) to allow the callback be executed before previously registered callbacks of this type.

`CallbackInterface` class provides following class methods:

`register_pre_callback`	registers *PRE* callback
`register_post_callback`	registers *POST* callback
`register_exc_callback`	registers *EXC* callback for purpose of recovering from execution errors
`register_interactive_prompt_callback`	registers callbacks called by the client context.

Let's look again at the last example:

```
 1:  from ipalib.plugins.user import user_add
 2:
 3:  def override_default_shell_cb(self, ldap, dn.
 4:                                entry_attrs, attrs_list,
 5:                                *keys, **options):
 6:      if 'loginshell' in entry_attrs:
 7:          default_shell = [self.api.Object.user.params['loginshell'].default]
 8:          if entry_attrs['loginshell'] == default_shell:
 9:              entry_attrs['loginshell'] = [u'/dev/null']
10:
11:  user_add.register_pre_callback(override_default_shell_cb)
```

This extension defines a pre-processing callback that accepts number of arguments:

ldap	reference to the back end to store and retrieve the object's data
dn	reference to the object data in LDAP
entry_attrs	arguments and options of the command and their values as a dictionary. All values in *entry_attrs* will be used for communicating with LDAP store, thus replacing values should be done with care. For details please see Python LDAP module documentation
attrs_list	list of all attributes we intend to fetch from the back end

keys arguments of the command

options all other unidentified parameters passed to the method

Arguments of a post-processing callback, *POST*, are slightly different. As action is already performed and the attributes of the entry are fetched back from the back end, there is no need to provide `attrs_list`:

```
1:  from ipalib.plugins.user import user_add
2:  def verify_shell_cb(self, ldap, dn. entry_attrs,
3:                      *keys, **options):
4:      if 'loginshell' in entry_attrs:
5:          default_shell = [self.api.Object.user.params['loginshell'].default]
6:          if entry_attrs['loginshell'] == default_shell:
7:              # report that default shell is assigned
8:
9:  user_add.register_post_callback(verify_shell_cb)
```

Execution error callback, *EXC*, has following signature:

```
1:  def user_add_error_cb(self, args, options, exc,
2:                        call_func, *call_args, **call_kwargs):
3:      return
```

where arguments have following meaning:

args arguments of the original method

options options of the original method

exc exception object thrown by a *call_func*

call_func function that was called by the method and caused the error of execution. In case of LDAP-based methods this is often `ldap.add_entry()` or `ldap.modify_entry()`, or a similar function

call_args first argument passed to the *call_func*

call_kwargs remaining arguments of *call_func*

Finally, interactive prompt callback receives *kw* argument which is a dictionary of all arguments of the command.

All callbacks are supplied with a reference to the method instance, `self`, unless the callback itself has an attribute called '`im_self`'. As can be seen in callback examples, self reference recursively provides access to the whole FreeIPA API structure.

This approach gives complete control of existing FreeIPA methods without deep dive into details of LDAP programming even if the framework allows such a deep dive.

Web UI

FreeIPA framework has two major client applications: Web UI and command line-based client tool, `ipa`. Web UI communicates with a FreeIPA server running WSGI application that accepts JSON-formatted requests and translates them to calls to FreeIPA plugins.

A following code in `install/share/ui/wsgi.py` defines FreeIPA web application:

```
 1:  from ipalib import api
 2:  from ipalib.config import Env
 3:  from ipalib.constants import DEFAULT_CONFIG
 4:
```

```
 5:  # Determine what debug level is configured. We can only do this
 6:  # by reading in the configuration file(s). The server always reads
 7:  # default.conf and will also read in `context'.conf.
 8:  env = Env()
 9:  env._bootstrap(context='server', log=None)
10:  env._finalize_core(**dict(DEFAULT_CONFIG))
11:
12:  # Initialize the API with the proper debug level
13:  api.bootstrap(context='server', debug=env.debug, log=None)
14:  try:
15:      api.finalize()
16:  except StandardError, e:
17:      api.log.error('Failed to start IPA: %s' % e)
18:  else:
19:      api.log.info('*** PROCESS START ***')
20:
21:      # This is the WSGI callable:
22:      def application(environ, start_response):
23:          if not environ['wsgi.multithread']:
24:              return api.Backend.session(environ, start_response)
25:          else:
26:              api.log.error("IPA does not work with the threaded MPM, use the pre-fo
```

At line 13 we set up FreeIPA framework with server context. This means plugins are loaded and initialized from following locations:

- `ipalib/plugins/` -- general FreeIPA plugins, available for all contexts
- `ipaserver/plugins/` -- server-specific plugins, available in '`server`' context

With `api.finalize()` call at line 15 FreeIPA framework is locked down and all components provided by plugins are registered at `api` name spaces: `api.Object`, `api.Method`, `api.Command`, `api.Backend`.

At this point, `api` name spaces become usable and our WSGI entry point, defined on lines 22 to 26 can access `api.Backend.session()` to generate response for WSGI request.

Web UI itself is written in JavaScript and utilizes JQuery framework. It can be split into three major parts:

communication	tools defined in `ipa.js` to allow talking with FreeIPA server using AJAX requests and JSON formatting
presentation	tools in `facet.js`, `entity.js`, `search.js`, `widget.js`, `add.js`, and `details.js` to give basic building blocks of Web UI
objects	actual implementation of Web UI for FreeIPA objects (user, group, host, rule, and other available objects registered at `api.Object` by the server side)

The code of these JavaScript files is loaded in `index.html` and kicked into work by `webui.js` where main navigation and document's `onready` event handler are defined. In addition, `index.html` imports `extension.js` file where all extensions to Web UI can be registered or referenced. As `extension.js` is loaded after all other Web UI JavaScript files but before `webui.js`, it can already use all tools of the Web UI.

The execution of Web UI starts with the call of `IPA.init()` function which does following:

1. Set up AJAX asynchronous communication via POST method using JSON format.
2. Fetches meta-data about FreeIPA methods available on the server using JSON format and makes them available as `IPA.methods`.
3. Fetches meta-data about FreeIPA objects available on the server using JSON format and makes them available as `IPA.objects`.

4. Fetches translations of messages used in the Web UI and makes them available as `IPA.messages`.

5. Fetches identity of the user running the Web UI, accessible as `IPA.whoami`.

6. Fetches FreeIPA environment specific for Web UI, accessible as `IPA.env`.

The communication with FreeIPA server is done using `IPA.command()` function. Commands created with `IPA.command()` can later be executed with `execute()` method. This separation of construction and actual execution allows to create multiple commands and combine them together in a single request. Batch requests are created with `IPA.batch_command()` function and command are added to them with `add_command()` method. In addition, FreeIPA Web UI allows to run commands concurrently with `IPA.concurrent_command()` function.

Web UI has following DOM structure:

	Container		
background	header	navigation	content
background-header	header-logo		
background-navigation	header-network-activity-indicator		
background-left	loggedinas		
background-right			

`Container` div is a top-level one, it includes background, header, navigation, content divs. These divs and their parts can be manipulated from the JavaScript code to represent the UI. However, FreeIPA gives an easier way to accomplish this.

Facets

Facet is a smallest block of FreeIPA Web UI. When facet is defined, it has name, label, link to an entity it is part of, and methods to create, show, load, and hide itself.

Entities

Entity is addressable group of facets. FreeIPA Web UI provides a declarative way of creating entities and defining their facets based on JavaScript's syntax. Following example is a complete definition of a netgroup facet:

```
 1:  IPA.netgroup = {};
 2:
 3:  IPA.netgroup.entity = function(spec) {
 4:      var that = IPA.entity(spec);
 5:      that.init = function(params) {
 6:          params.builder.search_facet({
 7:              columns: [
 8:                  'cn',
 9:                  'description'
10:              ]
11:          }).
12:          details_facet({
13:              sections: [
14:                  {
15:                      name: 'identity',
16:                      fields: [
17:                          'cn',
18:                          {
19:                              factory: IPA.textarea_widget,
```

```
20:                             name: 'description'
21:                         },
22:                         'nisdomainname'
23:                     ]
24:                 }
25:             ]
26:         }).
27:         association_facet({
28:             name: 'memberhost_host',
29:             facet_group: 'member'
30:         }).
31:         association_facet({
32:             name: 'memberhost_hostgroup',
33:             facet_group: 'member'
34:         }).
35:         association_facet({
36:             name: 'memberuser_user',
37:             facet_group: 'member'
38:         }).
39:         association_facet({
40:             name: 'memberuser_group',
41:             facet_group: 'member'
42:         }).
43:         association_facet({
44:             name: 'memberof_netgroup',
45:             associator: IPA.serial_associator
46:         }).
47:         standard_association_facets().
48:         adder_dialog({
49:             fields: [
50:                 'cn',
51:                 {
52:                     factory: IPA.textarea_widget,
53:                     name: 'description'
54:                 }
55:             ]
56:         });
57:     };
58:
59:     return that;
60: };
61:
62: IPA.register('netgroup', IPA.netgroup.entity);
```

This definition of a netgroup facet describes:

details facet — a facet named '`identity`' and three fields, `cn`, `description`, and `nisdomainname`. In addition, `description` field is a text area widget. This facet is used to display existing netgroup information.

association facets — number of facets, linking this one with others. In case of a netgroup, netgroups are linked to facet group `member` via different attributes. The definition also adds standard association facets defined in `entity.js`.

adder dialog — a dialog to create a new netgroup. The dialog has two fields: `cn` and `description` where `description` is again a text area widget.

Similarly to FreeIPA core framework, created entity needs to be registered to the Web UI via `IPA.register()` method.

In order to add new entity to the Web UI, one can use `extension.js`. This file in `/usr/share/ipa/html` is empty and provided specifically for this purpose.

As an example, let's define an entity 'Tank' corresponding to our aquarium tank:

```
 1:  IPA.tank = {};
 2:  IPA.tank.entity = function(spec) {
 3:      var that = IPA.entity(spec);
 4:      that.init = function(params) {
 5:          details_facet({
 6:              sections: [
 7:                  {
 8:                      name: 'identity',
 9:                      fields: [
10:                          'species', 'height', 'width', 'depth'
11:                      ]
12:                  }
13:              ]
14:          }).
15:          standard_association_facets().
16:          adder_dialog({
17:              fields: [
18:                  'species', 'height', 'width', 'depth'
19:              ]
20:          });
21:      };
22:  };
23:
24:  IPA.register('tank', IPA.tank.entity);
```

Command line tools

As an alternative to Web UI, FreeIPA server can be controlled via command-line interface provided by the `ipa` utility. This utility is operating under '`client`' context and looks even simpler than Web UI's `wsgi.py`:

```
1:  import sys
2:  from ipalib import api, cli
3:
4:  if __name__ == '__main__':
5:      cli.run(api)
```

`cli.run()` is the central running point defined in `ipalib/cli.py`:

```
 1:  # <cli.py code> ....
 2:  cli_plugins = (
 3:      cli,
 4:      textui,
 5:      console,
 6:      help,
 7:      show_mappings,
 8:  )
 9:
10:  def run(api):
```

```
11:      error = None
12:      try:
13:          (options, argv) = api.bootstrap_with_global_options(context='cli')
14:          for klass in cli_plugins:
15:              api.register(klass)
16:          api.load_plugins()
17:          api.finalize()
18:          if not 'config_loaded' in api.env:
19:              raise NotConfiguredError()
20:          sys.exit(api.Backend.cli.run(argv))
21:      except KeyboardInterrupt:
22:          print ''
23:          api.log.info('operation aborted')
24:      except PublicError, e:
25:          error = e
26:      except StandardError, e:
27:          api.log.exception('%s: %s', e.__class__.__name__, str(e))
28:          error = InternalError()
29:      if error is not None:
30:          assert isinstance(error, PublicError)
31:          api.log.error(error.strerror)
32:          sys.exit(error.rval)
```

As with WSGI, `api` is bootstraped, though with a client context and using global options from `/etc/ipa/default.conf`, and command line arguments. In addition to common plugins available in `ipalib/plugins`, `cli.py` adds few command-line specific classes defined in the module itself:

`cli` — a backend for executing from command line interface which does translation of command line option names, basic verification of commands and fallback to show help messages with `help` command, execution of the command, and translation of the output to command-line friendly format if this is defined for the command.

`textui` — a backend to nicely format output to stdout which handles conversion from binary to base64, prints text word-wrapped to the terminal width, formats returned complex values so that they can be easily understood by a human being.

```
>>> entry = {'name' : u'Test example', 'age' : u'100'}
>>> api.Backend.textui.print_entry(entry)
  age: 100
  name: Test example
```

`console` — starts interactive Python console with FreeIPA commands

`help` — generates help for every command and method of FreeIPA and structures it into sections according to the registered FreeIPA objects.

```
>>> api.Command.help(u'user-show')
Purpose: Display information about a user.
Usage: ipa [global-options] user-show LOGIN [options]

Options:
-h, --help  show this help message and exit
--rights    Display the access rights of this entry (requires --al
            ipa man page for details.
--all       Retrieve and print all attributes from the server. Aff
            command output.
--raw       Print entries as stored on the server. Only affects ou
```

format.

`show_mappings` displays mappings between command's parameters and LDAP attributes:

```
>>> api.Command.show_mappings(command_name=u"role-find")
Parameter : LDAP attribute
========= : ==============
name      : cn
desc      : description
timelimit : timelimit?
sizelimit : sizelimit?
```

Extending command line utility

Since `ipa` utility operates under client context, it loads all command plugins from `ipalib/plugins`. A simple way to extend command line is to drop its plugin file into `ipalib/plugins` on the machine where `ipa` utility is executed. Next time `ipa` is started, new plugin will be loaded together with all other plugins from `ipalib/plugins` and commands provided by it will be added to the `api`.

Let's add a command line plugin that allows to ping a server and measures round trip time:

```
 1:  from ipalib import frontend
 2:  from ipalib import output
 3:  from ipalib import _, ngettext
 4:  from ipalib import api
 5:  import time
 6:
 7:  __doc__ = _("""
 8:  Local extensions to FreeIPA commands
 9:  """)
10:
11:  class timed_ping(frontend.Command):
12:      __doc__ = _('Ping remote FreeIPA server and measure round-trip')
13:
14:      has_output = (
15:                              output.summary,
16:                              )
17:      def run(self):
18:          t1 = time.time()
19:          result = self.api.Command.ping()
20:          t2 = time.time()
21:          summary = u"""Round-trip to the server is %f ms.
22:  Server response is %s"""
23:          return dict(summary=summary % ((t2-t1)*1000.0, result['summary']))
24:
25:  api.register(timed_ping)
```

When this plugin code is placed into `ipalib/plugins/extend-cli.py` (name of the plugin file can be set arbitrarily), `ipa timed-ping` will produce following output:

```
$ ipa timed-ping
-------------------------------------------------------------------------------
Round-trip to the server is 286.306143 ms.
Server response is IPA server version 2.1.3GIT8a254ca. API version 2.13
-------------------------------------------------------------------------------
```

In this example we have created `timed-ping` command and overrode its `run()` method. Effectively, this command will only work properly on the client. If the client is also FreeIPA server (all FreeIPA servers are enrolled as FreeIPA clients), the same code will also be loaded by the server context and will be accessible to the Web UI as well, albeit its usefulness will be questionable as it will be measuring the round-trip to the server from the server itself.

File paths

Finally, it should be noted that depending on installed Python version and operating system, paths where plugins are loaded from may differ. Usually Python extensions are placed in `site-packages` Python sub-directory. In Fedora and RHEL distributions, this is `/usr/lib/python<version>/site-packages`. Thus, full path to `extend-cli.py` would be `/usr/lib/python<version>/site-packages/ipalib/plugins/extend-cli.py`.

On recent Fedora distribution, following paths are used:

Plugins	**Python module prefix**	**File path**
common	ipalib/plugins	/usr/lib/python2.7/site-packages/ipalib/plugins
server	ipaserver/plugins	/usr/lib/python2.7/site-packages/ipaserver/plugins
installer, updates	ipaserver/install/plugins	/usr/lib/python2.7/site-packages/ipaserver/install/plugins

Next table explains use of contexts in FreeIPA applications:

Context	**Application**	**Plugins**	**Description**
server	wsgi.py	common, server	Main FreeIPA server, server context
cli	ipa	common	Command line interface, client context
updates	ipa-ldap-updater	common, server, updates	LDAP schema updater

Platform portability

Originally FreeIPA was created utilizing packages available in Fedora and RHEL distributions. During configuration stages multiple system services need to be stopped and started again, scheduled to start after reboot and re-configured. In addition, when operating system utilizing security measures to harden the server setup, appropriate activities need to be done as well for preserving proper security contexts. As configuration details, service names, security features and management tools differ substantially between various GNU/Linux distributions and other operating systems, porting FreeIPA project's code to other environment has proven to be problematic.

When Fedora project has decided to migrate to systemd for services management, FreeIPA packages for Fedora needed to be updated as well, at the same time preserving support for older SystemV initialization scheme used in older releases. This prompted to develop a 'platformization' support allowing to abstract services management between different platforms.

FreeIPA 2.1.3 includes first cut of platformization work to support Fedora 16 distribution based on systemd. At the same time, there is an effort to port FreeIPA client side code to Ubuntu distributions.

Platform portability in FreeIPA means centralization of code to manage system-provided services, authentication setup, and means to manage security context and host names. It is going to be extended in future to cover other areas as well, both client- and server-side.

The code that implements platform-specific adaptation is placed under `ipapython/platform`. As of FreeIPA 2.1.3, there are two major "platforms" supported:

redhat Red Hat-based distributions utilizing SystemV init scripts such as Fedora 15 and RHEL6

fedora16 as name suggests, Fedora 16 and above, are supported by this platform module. It is based on `systemd` system management tool and utilizes common code in `ipapython/platform/systemd.py`. `fedora16.py` contains only differentiation required to cover Fedora 16-specific implementation of systemd use, depending on changes to Dogtag, Tomcat6, and 389-ds packages.

Each platform-specific adaptation should provide few basic building blocks:

AuthConfig class

`AuthConfig` class implements system-independent interface to configure system authentication resources. In Red Hat systems this is done with authconfig(8) utility.

`AuthConfig` class is nothing more than a tool to gather configuration options and execute their processing. These options then converted by an actual implementation to series of a system calls to appropriate utilities performing real configuration.

FreeIPA **expects** names of `AuthConfig`'s options to follow authconfig(8) naming scheme. From FreeIPA code perspective, the authentication configuration should be done with use of `ipapython.services.authconfig`:

```
 1:  from ipapython import services as ipaservices
 2:
 3:  auth_config = ipaservices.authconfig()
 4:  auth_config.disable("ldap").\
 5:              disable("krb5").\
 6:              disable("sssd").\
 7:              disable("sssdauth").\
 8:              disable("mkhomedir").\
 9:              add_option("update").\
10:              enable("nis").\
11:              add_parameter("nisdomain","foobar")
12:  auth_config.execute()
```

The actual implementation can differ. `redhat` platform module builds up arguments to authconfig(8) tool and on `execute()` method runs it with those arguments. Other systems will need to have processing of the arguments done as defined by authconfig(8) manual page. This is, perhaps, biggest obstacle on porting FreeIPA client side to the new platform.

PlatformService class

`PlatformService` class abstracts out an external process running on the system which is possible to administer: start, stop, check its status, schedule for automatic startup, etc.

Services are used thoroughly through FreeIPA server and client install tools. There are several services that are used especially often and they are selected to be accessible via Python properties of `ipapython.services.knownservices` instance.

To facilitate more expressive way of working with often used services, ipapython.services module provides a shortcut to access them by name via ipapython.services.knownservices.<service>. A typical code change looks like this:

```
from ipapython import services as ipaservices
```

```
....
-    service.restart("dirsrv")
-    service.restart("krb5kdc")
-    service.restart("httpd")
+    ipaservices.knownservices.dirsrv.restart()
+    ipaservices.knownservices.krb5kdc.restart()
+    ipaservices.knownservices.httpd.restart()
```

Besides expression change this also makes more explicit to platform providers access to what services they have to implement. Service names are defined in ipapython.platform.base.wellknownservices and represent definitive names to access these services from FreeIPA code. Of course, platform provider should remap those names to platform-specific ones -- for ipapython.platform.redhat provider mapping is identity.

Porting to a new platform may be hard as can be witnessed by this example: https://www.redhat.com/archives/freeipa-devel/2011-September/msg00408.html

If there is doubt, always consult existing providers. `redhat.py` is canonical -- it represents the code which was used throughout FreeIPA v2 development.

Enabling new platform provider

When support for new platform is implemented and appropriate provider is placed to `ipapython/platform/`, it is time to enable its use by the FreeIPA. Since FreeIPA is supposed to be rolled out uniformly on multiple clients and servers, best approach is to build and distribute software packages using platform-provided package management tools.

With this in mind, platform code selection in FreeIPA is static and run at package production time. In order to select proper platform provider, one needs to pass `SUPPORTED_PLATFORM` argument to FreeIPA's make process:

```
export SUPPORTED_PLATFORM=fedora16
# Force re-generate of platform support
rm -f ipapython/services.py
make version-update
make IPA_VERSION_IS_GIT_SNAPSHOT=no all
```

`version-update` target in FreeIPA top-level Makefile will re-create ipapython/services.py file based on the value of `SUPPORTED_PLATFORM` variable. By default this variable is set to `redhat`.

`ipapython/services.py` is generated using `ipapython/service.py.in`. In fact, there is only single line gets replaced in the latter file at the last line:

```
# authconfig is an entry point to platform-provided AuthConfig implementation
# (instance of ipapython.platform.base.AuthConfig)
authconfig = None

# knownservices is an entry point to known platform services
# (instance of ipapython.platform.base.KnownServices)
knownservices = None

# service is a class to instantiate ipapython.platform.base.PlatformService
service = None

# restore context default implementation  that does nothing
def restore_context_default(filepath):
    return

# Restore security context for a path
```

```
# If the platform has security features where context is important, implement your own
# version in platform services
restore_context = restore_context_default

# Default implementation of backup and replace hostname that does nothing
def backup_and_replace_hostname_default(fstore, statestore, hostname):
    return

# Backup and replace system's hostname
# Since many platforms have their own way how to store system's hostname, this method m
# implemented in platform services
backup_and_replace_hostname = backup_and_replace_hostname_default

from ipapython.platform.SUPPORTED_PLATFORM import *
```

As last statement imports everything from the supported platform provider, all exposed methods and variables above will be re-defined to platform-specific implementations. This allows to have FreeIPA framework use of these services separated from the implementation of the platform.

The code in ipapython/services.py is going to grow over time when more parts of FreeIPA framework become platform-independent.