freeIPA
identity | policy | audit

Extending the FreeIPA Server

Petr Viktorin & Petr Voborník

2014-02-17

Reasons to extend FreeIPA

- Adding a new attribute to an existing object type (e.g. host's OS name, user's employee ID) -- covered in these slides
- Adding a new object type (e.g. DHCP configuration)
- Adding/modifying functionality (e.g. default user login should be lowercased last name only)
- etc.

Adding a custom customer attribute to FreeIPA

- Need to do three steps:
  - Extending the schema (if attribute is not already there)
  - Adding an ipalib plugin
  - Adding a UI plugin

Extending the Schema

Extending the Schema: New attributeType

- Only needed when adding an *entirely new attribute* (not present in the schema already).
- Register a new OID for the attribute
- Add the new attribute with ldapmodify, e.g.:

```
dn: cn=schema
changetype: modify
add: attributeTypes
attributeTypes: ( 2.25.28639311321113238241701611583088740684.14.2.2
  NAME 'favoriteColorName'
  EQUALITY caseIgnoreMatch SUBSTR caseIgnoreSubstringsMatch
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  X-ORIGIN 'Extending FreeIPA' )
```

- See RHDS documentation for syntax reference

Extending the Schema: New objectClass

- Only needed when current objectClasses do not already contain the desired attributeType.
- Register a new OID for the objectClass
- Only include new attributes to the MAY clause
- Add the new objectClass with ldapmodify, e.g.:

```
dn: cn=schema
changetype: modify
add: objectclasses
objectclasses: ( 2.25.28639311321113238241701611583088740684.14.2.1
    NAME 'customPerson' SUP person
    STRUCTURAL
    MAY ( favoriteColorName )
    X-ORIGIN 'Extending FreeIPA' )
```

- See RHDS documentation for syntax reference

Schema: Replication and Upgrades

- Schema is replicated to all masters.

- Never modify existing objectClasses, always add new ones.
  Schema defined by FreeIPA is overwritten on FreeIPA upgrades.

- To update a custom attributeType or objectClass (for example to fix a mistake), add it again with ldapmodify. Make sure to use the same OID and name.

Adding custom objectClass for FreeIPA add commands

- Adding new objectClass for users:

```
ipa config-mod --addattr=ipaUserObjectClasses=customPerson
```

- Adding new objectClass for groups:

```
ipa config-mod --addattr=ipaGroupObjectClasses=newClass
```

- For modification commands (and for other object types), a plugin is needed.

Writing a Plugin

Adding an ipalib Plugin

- The UI/CLI does not automatically expose all attributes defined in the schema. Also, custom object classes are not automatically added to new/modified objects. For these tasks a plugin is needed.
- All ipalib plugins are located in `/usr/lib/python2.7/site-packages/ipalib/plugins/*.py`
- The plugin needs to be present on all FreeIPA servers *and* on all clients that will use the `ipa` command.
- After a plugin is added/changed on a server, Apache needs to be reloaded (`apachectl graceful`).

Exposing an Attribute in CLI

- Example plugin for exposing an attribute:

```
from ipalib.plugins import user
from ipalib.parameters import Str
from ipalib import _

user.user.takes_params = user.user.takes_params + (
    Str('favoritecolorname?',
        cli_name='color',
        label=_('Favorite color'),
    ),
)

user.user.default_attributes.append('favoritecolorname')
```

Parameter Names

```
from ipalib.plugins import user
from ipalib.parameters import Int
from ipalib import _

user.user.takes_params = user.user.takes_params + (
    Str('favoritecolorname?',
        cli_name='color',
        label=_('Favorite color'),
    ),
)

user.user.default_attributes.append('favoritecolorname')
```

- The main name is lowercased LDAP attribute name
- The cli_name is used for the CLI option name
- The label is used e.g. in CLI help

Specifying Multiple-Valued Attributes

- The tag after the name specifies whether a parameter accepts multiple values, and whether it is required.

```
user.user.takes_params = user.user.takes_params + (
    Str('favoritecolorname?',
        cli_name='color',
        label=_('Favorite color'),
    ),
)
```

- Use “?” (single optional value) or “*” (multiple values)
- Also available: no tag (single required value), and “+” (multiple values, at least one required)

Parameter Types

```
from ipalib.plugins import user
from ipalib.parameters import Int
from ipalib import _

user.user.takes_params = user.user.takes_params + (
        Int('employeenumber?',
            cli_name='number',
            label=_('Employee number'),
        ),
)
```

- Available parameter types: Str, Password, StrEnum, File, DNParam, ...
- See source (`ipalib/parameters.py`) for details

Simple Validation

- Common constraints can be specified in the parameter declaration
- Example – integer range validation:

```
user.user.takes_params = user.user.takes_params + (
        Int('employeenumber?',
            cli_name='number',
            label=_('Employee number'),
            minvalue=1,
            maxvalue=9999999,
        ),
)
```

- See source (`ipalib/parameters.py`) for others

Custom Validation

- A custom validator can be added as a Python function

```
def validate_color(ugettext, value):
    if value == 'magenta':
        return _("magenta is not acceptable")

user.user.takes_params = user.user.takes_params + (
    Str('favoritecolorname', validate_color,
        cli_name='color',
        label=_('Favorite color'),
    ),
)
```

```
$ ipa user-mod johnny --color=magenta
ipa: ERROR: invalid 'color': magenta is not acceptable
```

- Never use ipalib validators for security purposes (values can always be changed directly in LDAP)

Adding objectClass in a pre_callback

- In order to add a new objectClass to new and modified objects, we need to add pre_callbacks to the plugin.

```
def useradd_precallback(self, ldap, dn, entry, attrs_list,
                        *keys, **options):
    entry['objectclass'].append('customperson')
    return dn

user.user_add.register_pre_callback(useradd_precallback)

def usermod_precallback(self, ldap, dn, entry, attrs_list,
                        *keys, **options):
    if 'objectclass' not in entry.keys():
        old_entry = ldap.get_entry(dn, ['objectclass'])
        entry['objectclass'] = old_entry['objectclass']
    entry['objectclass'].append('customperson')
    return dn

user.user_mod.register_pre_callback(usermod_precallback)
```

Note: --setattr, --addattr, --delattr

- As an alternative to writing a plugin, if the attribute is included in the default objectClasses, we can use the --*attr options in the CLI without modifications to FreeIPA:

```
ipa user-mod --setattr=employeeNumber=123
```

- This is a quick lightweight solution for the CLI only.

Extending the Web UI

Web UI

- Has its own plugin system
- Plugins are written in JavaScript
- Plugins can add, remove, change or break the UI
  - Be careful!
- There is no plugin API atm, sorry.

Plugin name

- has to start with a letter and may contain only ASCII alphanumeric character, underscore _ and dash -
- used as a plugin directory name, AMD package name and main JavaScript file name

Plugin directory

- Plugins are located in `/usr/share/ipa/ui/js/plugins` directory.
- Each plugin has it's own subdirectory with the same name as plugin name
- e.g.: `/usr/share/ipa/ui/js/plugins/employeenumber`
- All plugin files should be in that subdirectory

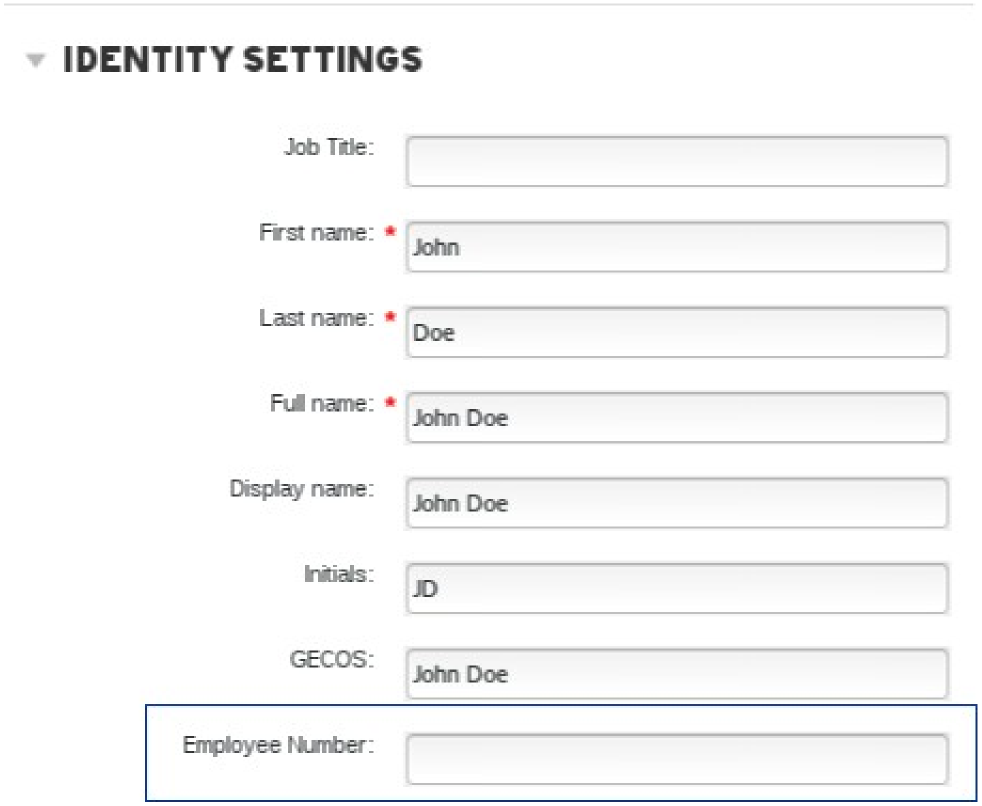
Main plugin file(module)

- Each plugin has to have a JavaScript file with the same name as plugin name, e.g., employeenumber.js
- This file is entry point of the plugin and is loaded by the plugin system
- It can point to other plugin files or files from core FreeIPA or different plugins. Module system [1] will take care of loading them.
- http://dojotoolkit.org/documentation/tutorials/1.8/modules/

Complete Example – add employee number field

```
define([
        'freeipa/phases',
        'freeipa/user'],
        function(phases, user_mod) {

// helper function
function get_item(array, attr, value) {

    for (var i=0,l=array.length; i<l; i++) {
        if (array[i][attr] === value) return array[i];
    }
    return null;
}

var emp_plugin = {};

// adds 'employeenumber' field into user details facet
emp_plugin.add_emp_number_pre_op = function() {

    var facet = get_item(user_mod.entity_spec.facets, '$type', 'details');
    var section = get_item(facet.sections, 'name', 'identity');
    section.fields.push({
        name: 'employeenumber',
        label: 'Employee Number'
    });
    return true;
};

phases.on('customization', emp_plugin.add_emp_number_pre_op);

return emp_plugin;
});
```

Example result


IDENTITY SETTINGS

Job Title:	
First name: *	John
Last name: *	Doe
Full name: *	John Doe
Display name:	John Doe
Initials:	JD
GECOS:	John Doe
Employee Number:	

Example detail – module definion

```
define([
        'freeipa/phases',
        'freeipa/user'],
        function(phases, user_mod) {

var emp_plugin = {};

return emp_plugin;
});
```

- Basic structure of a plugin
- Example has two dependencies:
  - FreeIPA core user module
  - FreeIPA core phases module

UI basics

- Ipalib/LDAP objects are called entities
- Pages are called facets
- Several types of facets, usually related to a type of FreeIPA command
  - Details – i.e., for output of user-show command
  - Search – i.e., for user-find
  - Association – to display member*
  - Nested search – special kind of search for nested objects – i.e., automount keys in automount maps

Specification objects

- UI is semi declarative – most entities, facets, facet content (widgets, field,...) are defined in specification objects
- Specification objects can be obtained from related modules from property named entity_spec or $NAME_spec if the module contains specs for several entities
- Use source code file search to locate particular module. Hint: UI code is in `install/ui/src/freeipa` dir
- You can load the module in browser dev tools, then examine the spec, e.g., by `require('freeipa/user')` call

Back to example

```
    var facet = get_item(user_mod.entity_spec.facets, '$type', 'details');
```

- 'freeipa/user' module was referenced in 'user_mod' variable
- Facets are defined in `facets` array
- We are interested in details facet which can be obtained by searching for object with `$type === 'details'`

Fields

- Object attributes are defined as fields
- Divided into sections
- Each section has a name
- In the example we wanted to get reference to 'identity' section:

```
var section = get_item(facet.sections, 'name', 'identity');
```

- Fields are defined in 'fields' array
- We can just append a new one:

```
section.fields.push({
     name: 'employeenumber',
     label: 'Employee Number'
});
```

Fields

- UI will get field metadata if the field is defined as a server side plugin param
- In such cases the declaraction can be simplified:

```
section.fields.push('employeenumber');
```

- You can specify a $type which controls which widget will be used - like a textarea or multivalued widget

```
{ $type: 'multivalued', name: 'mail' }
{ $type: 'textarea', name: 'description'}
```

- Note: In complex UIs facet or dialog might not have sections definition and instead it's defined in 'field' and 'widgets' separately with some related linking to each other

Hook to application lifecycle

- Web UI is started in several phases controlled by phases module
- Plugins authors who want to add/remove fields should be mostly interested in 'customization' phase – files are loaded, components are not registered nor created, specs objects are available for modification
- Register your own handler:

```
emp_plugin.add_emp_number_pre_op = function() {

    // my plugin code
    return true; // return some value or a promise in case of async operations
};

phases.on('customization', emp_plugin.add_emp_number_pre_op);
```

Complex plugins

- You can add complete entity pages or something completely different
- Consult source code for more information

freeIPA

identity | policy | audit