Managed Phones Plugin

You can provision phones through their configuration interface (init files, xml files, web GUI). However, as soon as you have more than a few phones provisioning becomes a repetitive and error-prone process. You can write a provisioning script or even provisioning UI. Or you can add support for the phone models to sipXconfig and let it provision your phones.

By adding support for your phone model you take advantage of sipXconfig open architecture: not only will other developers be able to add additional features to your particular phone configurator, but also every improvement to sipXconfig platform contributes new features to your phone plugin. What's more if your phone is supported by sipXconfig it will be managed with other phones, ATAs, and gateways. Out of the box, your phone configurator will support basic UI, configuration backup and templating mechanism. Later you will be able to add more advanced features such as intercom, speed dial, directory, time zone support etc.

Last but not least, adding support for your phone is fun. You get to use modern technologies and cool tools (Junit, Clover, Spring, Eclipse). You can try your hand at test driven development. You can participate in a well structured, active, open source project.

You do not have to contribute your plugin to SIPfoundry. However if you do, sipXconfig team will keep your code up-to-date with platform changes.

Prerequisites

Adding support will require basic experience in XML, Java and Linux. Before you start developing you need to know how to download, build and install sipXconfig and become familiar with sipXconfig web interface.

sipXconfig works best with phones that download (pull) their configuration through TFTP, FTP or HTTP. Other protocols and configuration methods (OMA /DM, SNMP) are also possible.

Introducing the Acme Phone

We'll create support for a fictitious phone called the Acme Phone. It supports 5 lines (also called registrations) and a handful of SIP related settings. It loads a single configuration file from a TFTP server and the format looks like a typical "INI" File.

Code Listing

Here is the final copy of all the source files for our sample phone that will be references throughout this document. You can use these files are starting points for your implementation.

1. AcmePhone.java - Java source code
2. AcmePhoneTest.java - Java unit test source code
3. expected-config - Java unit test source code
4. acmePhone/phone.xml - sipXconfig settings file format for phone settings
5. acmePhone/line.xml - sipXconfig settings file format for phone line settings
6. acme/acme-models.beans.xml - Plugin Definition Spring Framework file used to dynamically load your phone
7. acmePhone/config.vm - Configuration Velocity template for creating configuration file

Step 1: Capture Phone Settings - phone.xml

sipXconfig is a model driven configurator. Each configuration parameter is known as "setting". Your first task is to describe all the configuration parameters in a special XML format used by sipXconfig. This will ensure that sipXconfig knows how to generate configuration screens for your phones, and store settings values in the database.

Start by obtaining specifications of all the phone settings (or at least all the setings that you want to configure). You will need to describe each setting as one of the possible sipXconfig setting types and capture the results in an XML file.

If you're lucky enough to have an electronic copy of all the documented phone's settings for your phone, then maybe you can write a file parsing utilities to do the bulk of the conversion for you.

Q:How many settings do I need to support?

A: Some phones will ignore all settings set from the phone top when it loads any new settings from configuration file. Therefore if you do not support the bulk of the settings, the configuration system will not give your users the ability to preserve personal settings. For example, if you a user sets the headset volume to 5 and you do not capture headset volume, you may end up reseting the users headset volume each time you send new settings to the phone and not allow the user to preserve their headset volume.

If your phone can keep copies of local and provisioned settings, then you may want to concentrate on settings that are critical to a phones operation and that would need to be set for a large set of phones. For example, Outbound Proxy Transport Protocol: UDP v.s. TCP. If for some reason an admin needed to use one over the other.

Step 2: Capture Line Settings - line.xml

Phones that support multiple registrations will have a subset of settings that apply to each line or registration. For example, the authorization user id, the password (or passtoken). You need to pull these settings into a separate file named line.xml

Step 3: Implement Phone Object - AcmePhone.java

Start with a copy of AcmePhone.java, you'll want to make the following changes

Subclass from base phone class.

```
public class AcmePhone extends Phone \{
```

You'll want to select a new name for your phone and rename the Java source file appropriately.

It's important to know that a new instance of your phone class is instantiated each time sipXconfig generates a new configuration file, reboots a phone or displays settings screen in the web UI.

Define setting constants

```
   private static final String USER_ID_SETTING = "credential/userId";
```

The field name is arbitrary, but the path should correspond to path in `phone.xml` or `line.xml`. You only need to define settings that will need special processing, one of which is described in the next section.

Support for external lines

```
   @Override
   protected void setLineInfo(Line line, LineInfo info) \{
       line.setSettingValue(USER_ID_SETTING, info.getUserId());
...
   \}
```

```
   @Override
   protected LineInfo getLineInfo(Line line) \{
       LineInfo info = new LineInfo();
       info.setDisplayName(line.getSettingValue(DISPLAY_NAME_SETTING));
...
       return info;
   \}
```

Here we account for all the very basic line information the system would need to examine and create a registration for your phone. This information is important when creating external line registrations, but may have other uses in the future.

Load settings files

```
   @Override
   protected Setting loadSettings() \{
       return getModelFilesContext().loadModelFile("phone.xml", getBeanId());
   \}

   @Override
   protected Setting loadLineSettings() \{
       return getModelFilesContext().loadModelFile("line.xml", getBeanId());
   \}
```

This is required and typical for all phones. Here you would have an opportunity to alter the settings model, for each phone instance.

Line Runtime Defaults

Approach 1: Java annotations

```
    @Override
    public void initializeLine(Line line) \{
        line.addDefaultBeanSettingHandler(new AcmeLineDefaults(getPhoneContext()));
    \}
```

Here we delegate "filling-in" phone line settings for the administrator to `AcmeLineDefaults` so administrators do not have to fill them in themselves.

```
    public static class AcmeLineDefaults \{
        private Line m_line;
        AcmeLineDefaults(Line line) \{
            m_line = line;
        \}

        @SettingEntry(path = USER_ID_SETTING)
        public String getUserName() \{
            String userName = null;
            User user = m_line.getUser();
            if (user != null) \{
                userName = user.getUserName();
            \}
            return userName;
        \}
...
```

This can range from the address of the voicemail server to what the magic keyword is for intercom feature. Admins will always have the opportunity to inspect and override these settings from the web UI for each phone or any phone group.

The more you handle in this section, the better your "out of the box" experience is for your users. Go crazy: add speed-dial buttons to configured voicemail servers or park servers.

There are 2 ways of providing defaults for settings. In the first method you construct a Java class and use Java annotations to match settings to code. Here we obtain the user id from the system user and "fill-in" the user id setting for AcmePhone. This is used for presenting values in the web ui and when generating profiles.

This uses Java annotations to associate a setting to a method call. What's nice about this approach is that you can write plain old Java code to determine your default settings and also unit test your code by creating very few additional classes.

Approach 2: Implementing SettingValueHandler'

When the settings you want to influence match a particular pattern, you should investigate implementing SettingValueHandler interface. For example, all integer settings, or setting names that end with "_xyz"

Phone Runtime Defaults

Phone-wide settings (as opposed to line settings) have a similiar opportunity to supply defaults by implementing `initialize()` method.

Step 4: Create Plugin Descriptor - acme-models.beans.xml

In order for the system to load support for you new phone model, you must create this descriptor file. File name must match the following wildcard expression `*-models.bean.xml` to get loaded automatically by the Spring framework.

File Format

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
```

You may have noticed from the DTD reference that this is a Spring Framework file. You don't need to know Spring Framework, all that you should need is described below.

Phone Model(s)

```
<bean id="acmePhoneStandard" class="org.sipfoundry.sipxconfig.phone.PhoneModel">
  <property name="beanId" value="acmePhone"/>
  <property name="label" value="Acme"/>
  <property name="maxLineCount" value="5"/>
</bean>
```

This registers your phone with the system so users can select it as an available phone model. You may create as many of these <bean> elements as models you wish to distinguish. Every phone implementation needs at least one.

Values of note:

- id - unique identifier for one particular model of your phone. This must not change (unless you create a database patch beyond the scope of this document.)
- label - Name that appears from web UI to distinguish your phone to end user
- maxLineCount - Total number of registrations your phone can support. This is not the number of simultaneous calls, which is not captured here.

Phone Class

```
<bean id="acmePhone" class="org.sipfoundry.sipxconfig.phone.acme.AcmePhone" singleton="false"
  parent="abstractPhone">
</bean>
```

This registers your phone implementation so the system knows how to instantiate your code.

Values of note:

- id - unique identifier for your implementation
- singleton="false" - required are must always be false, this is to ensure a new instance of your phone is created to represent each phone in your system
- parent="abstractPhone" - injects all the typical services a phone uses.

Advanced Topic: Access to other settings or services

From this plugin descriptor file you can gain access to other parts of the system that are of interest to your phone. For example, if your phone could browse LDAP servers, you could get access to the LDAP settings

```
<bean id="acmePhone" ...>
  <property name="ldapManager" ref="ldapManager"/>
</bean>
```

'neoconf/src/org/sipfoundry/sipxconfig/phone/acme/AcmePhone.java'

```
...
public setLdapManager(LdapManager ldapManager) {
    m_ldapManager = ldapManager;
}
...
```

-- *MichalBielicki*
Q: How does one know what services are available or their names?
-- *Lazyboy*
A: No great way, but every part of the system is available from Spring so if it's configurable it's available somewhere.
Browse all *.beans.xml files. This also looks promising http://opensource.atlassian.com/confluence/spring/display/BDOC/Home

Step 5: Generating Profiles - config.vm

AcmePhone will generate its configation in the text file in the TFTP root directory. Although overkill for this simple example I will use Velocity to create a template to help generate a profile. You can implement any scheme you want including simply dumping setting values into a flat file using standard Java file IO. Velocity is a templating tool and is embedded into sipXconfig but it is not required to be used for configuration file generation. Its templating language is documented on the Velocity web site.

Here the first half of what my file looks like

'etc/unmanagedPhone/config.vm'

```
#foreach ($group in $phone.Settings.Values)
[$\{group.Name\}]
#foreach ($setting in $group.Values)
$\{setting.ProfileName\}=$!\{setting.Value\}
#end

...
```

This template will copy all settings directly into the configuration file. Because of the simplicity of this method the setting groups and settings must be organized exactly how the config file breaks its settings into INI sections. The is fine for generating profiles, but may not always be helpful for the user. This is fine for now, however if this changes then settings can be filtered before passing them to Velocity or Velocity can call filter methods from java.

Step 6: Creating a unit test - AcmePhoneTest.java

'AcmePhoneTest.java'

```
...
   public void testGenerateTypicalProfile() throws Exception {
       AcmePhone phone = new AcmePhone();

       // call this to inject dummy data

       PhoneTestDriver.supplyTestData(phone);

       StringWriter actualWriter = new StringWriter();
       phone.generateProfile(actualWriter);
       InputStream expectedProfile = getClass().getResourceAsStream("expected-config");
       String expected = IOUtils.toString(expectedProfile);
       expectedProfile.close();

       assertEquals(expected, actualWriter.toString());
   }
...
```

This instructs your phone to generate a configuration into memory, then compares the configuration to what you expect it to be. Here we use a utility class `PhoneTestDriver` to create sample data but may create your own. Every single setting is compared, so means the unit test will have to be updated quite regularly, however if there was a problem with any of the settings, this is the best way for you and others find the issue.

Running unit test

To run my test, I go to unix or DOS prompt and run the following commands:

```
cd sipXconfig/neoconf
ant -Dtest.name=AcmePhone default test
```

If you get an error, you'll find results in `test-results/TEST-org.sipfoundry.sipxconfig.phone.acme.AcmePhoneTest.txt`

All the unit tests can run from eclipse if you've set up your eclipse environment accordingly.

Step 6: Submitting to sipXconfig project

Before submitting your code you need to run a couple of checks.

- Run coding style checker to ensure source code passed the coding conventions (standard java coding standards: http://java.sun.com/docs/codeconv/)

```
ant style
```

- Run the clover coverage tool. Pingtel has a clover RPM and licence file if you don't already have clover installed. Clover's demo license should last 30 days until you can obtain one by emailing the sipx-dev list.

```
ant -Dwith.clover clean default test-all clover.viewer
```

Advanced Features

Speed dials

Support for speed dials allows phone to take advantage of Speed Dial UI screen available in User Portal.

To implement speed dial support phone plugin needs to retrieve the speed dial from PhoneContext and use it to generate phone configuration profile.

```
SpeedDial speedDial = getPhoneContext().getSpeedDial(this);
```

SpeedDial object contains a list of buttons. If phone supports BLF it should check if the button is "monitored" and treat it appropriately.
See LG-Nortel plugin for an example.

Phonebooks

sipXconfig admin can configure phonebooks for end user. To take advantage of the phonebook feature phone plug-in needs to retrieve the list of entries for the phone and then use the list when generating configuration profiles.

```
Collection<PhonebookEntry> entries = getPhoneContext().getPhonebookEntries(this);
```

See Polycom plugin provides a good example.

Others

- DST parameters
- NTP address and other network paramters
- Model variations and conditional settings
- Profile delivery

Contributed Phone Support To Date

- Niels Ohlmeier added support for the Snom Phones
- Hannu Strang added support for the Grandstream Phones
- Hannu Strang also added support for Cisco Phones
- Michal Bielicki added support for the Hitachi wireless Phones
- Michal Bielicki also upgraded support for the Snom Phones
- Douglas Hubler added support for Polycom Phones
- Sen Heng added support for the Linksys Phones
- Sen Heng also added support for the New Cisco Phones
- *Your name here and your phone here*