Contents

Introduction

The XML dialplan is the default dialplan used by FreeSwitch. XML is easily edited by hand without requiring special tools, other than a text editor.

Dialplans are used to route a dialed call to its appropriate endpoint, which can be a traditional extension, voicemail, interactive voice response (IVR) menu or other compatible application. Dialplans are extremely flexible.

Dialplans can be separated into 'contexts' allowing you to have separate dialplans for different types of calls. Calls can be handed off to different contexts as well. For example, to ease security concerns, you might have two dialplans, one that handles calls originating from the public phone network (PSTN) and one that handles calls originating from internal extensions. The FreeSwitch sample configuration files use this method, forcing an incoming PSTN call through some additional scrutiny before being handed off to the internal dialplan. Another use of a dialplan context allows you to share a single PBX with multiple tenants in an office building. Since each tenant will likely have their own set of (and often conflicting) extensions, voice menus, etc. it makes sense to separate tenants into their own independent dialplans to ease configuration and maintenance.

XML dialplans use the standardized Perl Regular Expression matching on fields, which decreases the "learning curve" of creating and maintaining dialplans for experienced Perl users. Understanding this documentation requires an understanding of regular expressions.

The 10,000 Foot View

This overview uses the Sofia SIP driver as the source and destination for calls handled by the dialplan. Other drivers have similar mechanisms.

When a call arrives at the FreeSWITCH, Sofia is the first responder. It gathers information about the call, and decides which dialplan to invoke.

Sofia passes information about the call in 'channel variables' to the appropriate dialplan, called a dialplan 'context'.

The dialplan looks at the channel variables to determine how to handle the call. It does this by sifting through a list of possible match 'conditions' set forth in the dialplan. If a match is found, a matching set of dialplan 'applications' are run to handle the call. Applications can include transferring the control of the call to another dialplan, bridging the call to an extension, voicemail system, IVR, etc. or any other application integrated into FreeSWITCH.

The key to calling your dialplan is the following XML element in your Sofia configuration file.

```
<param name="context" value="public"/>
```

Dialplan Configuration Directory Structure

The FreeSWITCH example configuration stores dialplans in the `conf/dialplans` directory, with each context stored in a subdirectory beneath it. This is the recommended configuration.

For example, this sample structure houses a dialplan for a simple PBX:

```
freeswitch/
    conf/
        dialplans/
            public.xml
            public/
                00_security_screen.xml
                10_inbound_sip-bandwidth-r-us.xml
                29_inbound_sip-super-call.xml
                ...
            default.xml
            default/
                00_feature_codes.xml
                05_voicemail_access.xml
                20_extension_2001.xml
                20_extension_2002.xml
                ...
```

A few things to notice:

- This dialplan structure is similar to the example configuration that ships with FreeSWITCH. It is a good starting point to customize your own PBX, as we have done here.
- Any `.xml` file placed in the `dialplans` directory will be picked up when FreeSWITCH starts. This isn't magic, this is how the default configuration is loaded.
- `public.xml` contains all the common configuration information for the public context, then includes all the XML files in the public directory.
- `default.xml` contains all the common configuration information for the default context, then includes all the XML files in the default directory.
- Files are included in alphabetical order, so beginning file names with a number is a good practice to ensure the bits and pieces of your dialplan are processed in the right order.

Anatomy of the XML Dialplan

There are several elements used to build an XML dialplan. In general, the dialplan groups logically similar functions and calling activities into a 'context'. Within a context are extensions, each with 'condition' rules and

associated 'actions' to perform when the condition rules match.

The following is a sample dialplan to illustrate these concepts. We have left out the XML "wrapper" to help make the basic concepts more clear:

```
<context name="example">
    <extension name="500">
        <condition field="destination_number" expression="^500$">
            <action application="bridge" data="user/500"/>
        </condition>
    </extension>

    <extension name="501">
        <condition field="destination_number" expression="^501$">
            <action application="bridge" data="user/501"/>
            <action application="answer"/>
            <action application="sleep" data="1000"/>
            <action application="bridge" data="loopback/app=voicemail:default ${domain_name
        </condition>
    </extension>
</context>
```

Each rule is processed in order until you reach the action tag which tells FreeSWITCH what action to perform. You are not limited to only one condition or action tag for a given extension.

In our above example, a call to extension 501 rings the extensions. If the user does not answer, the second action answers the call, and following actions delay for 1000 milliseconds (which is 1 second) and connect the call to the voicemail system.

Context

Contexts are a logical grouping of extensions. You may have multiple extensions contained within a single context.

The context tag has a required parameter of 'name'. There is one reserved name, `any`, which matches any context. The name is used by incoming call handlers (like the [Sofia] SIP driver) to select the dialplan that runs when it needs to route a call. There is often more than one context in a dialplan.

A fully qualified context definition is shown below. Typically you'll not need all the trimmings, but they are shown here for completeness.

```
<?xml version="1.0"?>
    <document type="freeswitch/xml">
        <section name="dialplan" description="Regex/XML Dialplan">
        <!-- the default context is a safe start -->
            <context name="default">
                <!-- one or more extension tags -->
            </context>
            <!-- more optional contexts -->
        </section>
    </document>
```

Extensions

Extensions are destinations for a call. This is the meat of FreeSWITCH routing dialed numbers. They are given a name and contain a group of conditions, that if met, will execute certain actions.

A 'name' parameter is required: It must be a unique name assigned to an extension for identification and later use.

For example:

```
<extension name="Your extension name here">
    <condition(s)...
        <action(s) .../>
    </condition>
</extension>
```

NOTE: Typically when an extension is matched in your dialplan, the corresponding actions are performed and dialplan processing stops. An optional `continue` parameter allows your dialplan to continue running.

```
<extension name="500" continue="true">
```

Conditions

Dialplan conditions are typically used to match a destination number to an extension. They have, however, much more power than may appear on the surface.

FreeSWITCH has a set of built-in variables used for testing. In this example, the built-in variable `destination_number` is compared against the regular expression `^500$`. This comparison is 'true' if <destination_number> is set to `500`.

```
<extension name="500">
    <condition field="destination_number" expression="^500$">
        <action application="bridge" data="user/500"/>
    </condition>
</extension>
```

Each condition is parsed with the Perl Compatible Regular Expression library. (go here for PCRE syntax information).

If a regular expression contains any terms wrapped in parentheses, and the expression matches, the variables $1,$2..$N will be set to the matching contents within the parenthesis, and may be used in subsequent action tags within this extension's block.

For example, this simple expression matches a four digit extension number, and captures the last two digits into $1.

```
<condition field="destination_number" expression="^\d\d(\d\d)$">
    <action application="bridge" data="sofia/internal/$1@example.com"/>
</condition>
```

A destination number of 3425 would set $1 to 25 and then bridge the call to the phone at `25@example.com`

Multiple Conditions (Logical AND)

You can emulate the logical AND operation available in many programming languages using multiple conditions.

When you place more than one condition in an extension, *all* conditions must match before the actions will be executed.

For example, this block will only execute the actions if the destination number is 500 *AND* it is Sunday.

```
<condition field="destination_number" expression="^500$"/>
<condition wday="1">
    action(s)...
</condition>
```

Keep in mind that you must observe correct XML syntax when using this structure. Be sure to close all conditions *except the last one* with `/>`. The last condition contains the final actions to be run, and is closed on the line after the last action.

By default, if any condition is false, FreeSWITCH will move on to the anti-actions or the next extension without even evaluating any more conditions.

Nested Conditions

Recently added. See http://jira.freeswitch.org/browse/FS-4935

The new attribute in the nested conditions is "require-nested", it is true or false and indicates if the nested conditions must evaluate positive to execute the other actions in the top level condition the attribute is placed in.

So below if "require-nested" is true which is the default even when not specified, the sub-condition must either be true or have break=never set in the same fashion as top-level extensions. Basically each level of nested conditions are parsed with the identical method that an extension is parsed only instead of continue you use the "require-nested" logic described above.

If require-nested is false then no matter what happens in the nested condition, even if it fails completely, it will pass on to the next actions in the parent.

```
<extension name="nested_example">
 <condition field="destination_number" expression="^1234$" require-nested="true">

   <condition field="destination_number" expression="3">
     <action application="playback" data="foo.wav" />
   </condition>

   <action application="playback" data="bar.wav" />

 </condition
</extension>
```

Basically with this combination of logic you should be able to do one main condition with "require-nested=false" then a series of nested conditions each implementing the logical AND.

```
      <extension name="nested_example">
        <condition field="destination_number" expression="^1234$" require-nested="false">

          <condition field="f1" expression="e1"/>
          <condition field="f2" expression="e2"/>
          <condition field="f3" expression="e3"/>
          <condition field="destination_number" expression="3" break="never">
            <action application="playback" data="foo.wav" />
          </condition>

          <action application="playback" data="bar.wav" />

        </condition
      </extension>
```

Nested Condition Caveats and Notes

The first thing to keep in mind is that when the parsing finds a condition with nested child conditions it will always parse the children first! That means you cannot do this:

```
<extension name="nested_example">
  <condition field="destination_number" expression="^1234$" require-nested="false">
    <action application="log" data="INFO I'm before the nested conditions..."/>

    <condition field="${foo1}" expression="bar1" break="never">
      <action application="log" data="INFO foo1 is bar1" />
    </condition>

    <action application="log" data="INFO I'm in between the nested conditions..."/>

    <condition field="${foo2}" expression="bar2" break="never">
      <action application="log" data="INFO foo2 is bar2" />
    </condition>

    <action application="log" data="INFO I'm after the nested conditions..."/>

  </condition
</extension>
```

It will not do what you expect. Instead, it will execute the lines in this order:

```
<action application="log" data="INFO foo1 is bar1" />
<action application="log" data="INFO foo2 is bar2" />
<action application="log" data="INFO I'm before the nested conditions..."/>
<action application="log" data="INFO I'm in between the nested conditions..."/>
<action application="log" data="INFO I'm after the nested conditions..."/>
```

So, remember: children first! Their actions/anti-actions will get processed and added to the task list before the parents' will.

Other important note: $1, $2, etc. capture DOES work:

```
   <extension name="Local_Extension">
     <condition field="destination_number" expression="^($${extension_regex})$" require-nested="f
       <condition field="caller_id_number" expression="^(.*)$">
         <action application="log" data="WARNING Inside nested condition, CID Num is $1"/>
       </condition>
       <action application="log" data="NOTICE Can you see me?"/>
```

```
<condition field="caller_id_name" expression="^(.*)$">
  <action application="log" data="WARNING Inside nested condition, CID Name is $1"/>
</condition>
<action application="log" data="WARNING dest num is $1"/>
...
```

The above will work just fine. Keep these two things in mind and you'll be doing all sorts of interesting things with your dialplans!

Multiple Conditions (Logical OR, XOR)

It is possible to emulate the logical OR operation available in many programming languages, using multiple conditions. In this situation, if one of the conditions matches, the actions are executed.

For example, this block executes its actions if the destination number is 501 *OR* the destination number is 502.

```
<condition field="destination_number" expression="^501|502$">
    action(s)...
</condition>
```

This method works well if your OR condition is for the same field. However, if you need to use two or more different fields then use the new **regex** syntax added on November 1, 2011. It looks like this:

```
<condition regex="all|any|xor">
  <regex field="some_field" expression="Some Value"/>
  <regex field="another_field" expression="^Another\s*Value$"/>
  <action(s) ...>
  <anti-action(s)...>
</condition>
```

The regex condition can have one of three values:

- all ? is equivalent to a logical AND operation. *All* of the expressions contained in the condition **must** be true for the actions to be taken.
- any ? is the equivalent to a logical OR operation. *Any* of the expressions contained in the condition **may** be true for the actions to be taken.
- xor ? is the equivalent to a logical XOR operation. Only **one** of the expressions can be true for the actions to be taken.

Actions and anti-actions are executed like they would be in a standard *condition* block.

For example, this block checks the caller ID name and number, and executes the actions if the name is "Some User" **OR** the number is 1001:

```
<extension name="Regex OR example 1" continue="true">
  <condition regex="any">
    <!-- If either of these is true then the subsequent actions are added to execute lis
    <regex field="caller_id_name" expression="Some User"/>
    <regex field="caller_id_number" expression="^1001$"/>
    <action application="log" data="INFO At least one of the conditions matched!"/>
    <!-- If *none* of the regexes is true then the anti-actions are added to the execute
    <anti-action application="log" data="WARNING None of the conditions matched!"/>
  </condition>
```

```
</extension>
```

This method makes it easier to match the caller's name *OR* caller ID number and execute actions when either is true.

A slightly more advanced use of this method is demonstrated below. If the caller's name is "Michael S Collins" **OR** the caller ID is 1002, 3757, or 2816, the variable *calling_user* is set to "mercutioviz". If neither is true, then the *calling_user* variable is set to "loser". After playing the welcome message, a custom message is played based on the calling_user variable.

```
<extension name="Regex OR example 2" continue="true">
  <condition regex="any" break="never">
    <regex field="caller_id_name" expression="^Michael\s*S?\s*Collins"/>
    <regex field="caller_id_number" expression="^1001|3757|2816$"/>
    <action application="set" data="calling_user=mercutioviz" inline="true"/>
    <anti-action application="set" data="calling_user=loser" inline="true"/>
  </condition>

  <condition>
    <action application="answer"/>
    <action application="sleep" data="500"/>
    <action application="playback" data="ivr/ivr-welcome_to_freeswitch.wav"/>
    <action application="sleep" data="500"/>
  </condition>

  <condition field="${calling_user}" expression="^loser$">
    <action application="playback" data="ivr/ivr-dude_you_suck.wav"/>
    <anti-action application="playback" data="ivr/ivr-dude_you_rock.wav"/>
  </condition>
</extension>
```

The next example shows how to create an XOR (exclusive-or) condition block, which is true when *only one* of the conditions is true.

```
<extension name="Regex XOR example 3" continue="true">
  <condition regex="xor">
    <!-- If only one of these is true then the subsequent actions are added to execute l
    <regex field="caller_id_name" expression="Some User"/>
    <regex field="caller_id_number" expression="^1001$"/>
    <action application="log" data="INFO Only one of the conditions matched!"/>
    <!-- If *none* of the regexes is true then the anti-actions are added to the execute
    <anti-action application="log" data="WARNING None of the conditions matched!"/>
  </condition>
</extension>
```

Another example shows how to ensure that all expressions match before executing actions, otherwise the anti-actions will be executed. In this case, the SIP gateway **must** be the default provider, **and** it must be an emergency call, **and** the auto-answer option must be enabled and stored in the database:

```
<condition regex="all">
 <regex field="${sip_gateway}" expression="^${default_provider}$"/>
 <regex field="${emergency_call}" expression="^true$"/>
 <regex field="${db(select/emergency/autoanswer)}" expression="^1$"/>

 <!-- the following actions get executed if all regexes PASS -->
  <action application="set" data="call_timeout=60"/>
  <action application="set" data="effective_caller_id_name=${regex(${caller_id_name}|^Eme
  <action application="set" data="autoanswered=true"/>
```

```
    <action application="bridge" data="user/1000@${domain_name},sofia/gateway/1006_7217/${m

   <!-- the following anti-actions are executed if any of the regexes FAIL -->
    <anti-action application="set" data="effective_caller_id_name=${regex(${caller_id_name}
    <anti-action application="set" data="call_timeout=30"/>
    <anti-action application="set" data="autoanswered=false"/>
    <anti-action application="bridge" data="user/1000@${domain_name},sofia/gateway/1006_721
  </condition>
```

Note: If you capture data in any of the regular expressions using the ( ) operators, only the data from the last capture encountered will be considered when expanding the capture (using the $n syntax) in any following actions. [*What does this mean?:* In addition, you can set DP_REGEX_MATCH_1 .. DP_REGEX_MATCH_N to preserve captures into arrays.]

Complex Condition/Action Rules

Here is a more complex example, performing time-based routing for a support organization. The user dials extension 1100. The actual support extension is 1105 and is staffed every day from 8am to 10pm, except Friday, when it is staffed between 8am and 1pm. At all other times, calls to 1100 are sent to the support after-hours mailbox.

Note that `break="on-false"` is the default.

```
<extension name="Time-of-day-tod">
    <!--if this is false, FreeSWITCH skips to the next *extension*.-->
    <condition field="destination_number" expression="^1100$" break="on-false"/>

    <!--Don't bother evaluating the next condition set if this is true.-->
    <condition wday="6" hour="8-12" break="on-true">    <!--Fri, 8am-12:59pm-->
        <action application="transfer" data="1105 XML default"/>
    </condition>

    <condition wday="1-5" hour="8-21" break="on-true">   <!--Sunday-Thursday, 8am-9:59pm--
        <action application="transfer" data="1105 XML default"/>
    </condition>

    <condition> <!--this is a catch all, sending the call to voicemail at all other times.
        <action application="voicemail" data="default ${domain} 1105"/>
    </condition>
</extension>
```

In this example, we use the `break=never` parameter to cause the first condition to 'fall-through' to the next condition no matter if the first condition is true or false. This is useful to set certain flags as part of extension processing. This example sets the variable `begins_with_one` if the destination number begins with 1.

```
<extension name="break-demo">
 <!-- because break=never is set, even when the destination does not begin
      with 1, we skip the action and keep going -->
 <condition field="destination_number" expression="^1(\d+)$" break="never">
   <action application="set" data="begins_with_one=true"/>
 </condition>

 <condition field="destination_number" expression="^(\d+)$">
    ...other actions that may query begins_with_one...
```

```
 </condition>
</extension>
```

Asterisk Patterns

In addition to PCRE FreeSWITCH also supports Asterisk Patterns any expression that starts with an underscore (_) will use asterisk pattern matching. please note that some patterns may require escaping, such as the "*"

See Mod_dialplan_asterisk

```
<extension name="_NXXXXXXXXX_asterisk">
  <condition field="destination_number" expression="_(NXXXXXXXXX)">
    <action application="bridge" data="sofia/internal/$1@example.com"/>
  </condition>
</extension>

<extension name="_*XX_with_escaping">
  <condition field="destination_number" expression="_(\*XX)(.)">
    <action application="log" data="ERR captured $1 ~~~ $2"/>
      <action application="answer"/>
      <action application="playback" data="tone_stream://path=${base_dir}/conf/tetris.ttm
    </condition>
</extension>
```

Variables

Condition statements can match against channel variables, or against an array of built in variables.

Built-In Variables

The following variables, called 'caller profile fields', can be accessed from condition statements directly:

- **context** Why can we use the context as a field? Give us examples of usages please.
- **rdnis** Redirected Number, the directory number to which the call was last presented.
- **destination_number** Called Number, the number this call is trying to reach (within a given context)
- **dialplan** Name of the dialplan module that are used, the name is provided by each dialplan module. Example: XML
- **caller_id_name** Name of the caller (provided by the User Agent that has called us).
- **caller_id_number** Directory Number of the party who called (caller) -- can be masked (hidden)
- **ani** Automatic Number Identification, the number of the calling party (caller) -- cannot be masked

- **aniii** The type of device placing the call ANI2
- **uuid** Unique identifier of the current call? (looks like a GUID)
- **source** Name of the FreeSWITCH module that received the call (e.g. PortAudio)
- **chan_name** Name of the current channel (Example: PortAudio/1234). Give us examples when this one can be used.
- **network_addr** IP address of the signaling source for a VoIP call.
- **year** Calendar year, 0-9999
- **yday** Day of year, 1-366
- **mon** Month, 1-12 (Jan = 1, etc.)
- **mday** Day of month, 1-31
- **week** Week of year, 1-53
- **mweek** Week of month, 1-6
- **wday** Day of week, 1-7 (Sun = 1, Mon = 2, etc.) or "sun", "mon", "tue", etc.
- **hour** Hour, 0-23
- **minute** Minute (of the hour), 0-59
- **minute-of-day** Minute of the day, (1-1440) (midnight = 1, 1am = 60, noon = 720, etc.)
- **time-of-day** Time range formatted: hh:mm[:ss]-hh:mm[:ss] (seconds optional) Example: "08:00-17:00"
- **date-time** Date/time range formatted: YYYY-MM-DD hh:mm[:ss]~YYYY-MM-DD hh:mm[:ss] (seconds optional, note tilde between dates) Example: 2010-10-01 00:00:01~2010-10-15 23:59:59

For example:

```
<condition field="network_addr" expression="^192\.168\.1\.1$"/>  <!-- network address=192.
<condition mon="2">   <!-- month=February -->
```

Caller Profile Fields vs. Channel Variables

One thing that may seem confusing is the distinction between a caller profile field (the built-in variables) and a channel variable.

Caller profile fields are accessed like this:

```
<condition field="destination_number" attributes...>
```

While channel variables are accessed like this:

```
<condition field="${sip_has_crypto}" attributes...>
```

Please take note of the **${variable_name}** syntax. Channel variables may also be used in action statements.

In addition, API functions can be called from inside a condition statement to provide dynamic data.

For example, you can use the **cond** API:

```
<condition field="${cond(${my_var} > 12 ? YES : NO)}" expression="^YES$">
    <action application="log" data="INFO ${my_var} is indeed greater than 12"/>
</condition>
```

This example tests ${my_var}. If it is more than 12, "YES" is returned. Otherwise "NO" is returned. The condition tests the results for "YES" and logs the resulting message to the FreeSWITCH log.

Availability of Variables

Asterisk users must read!

The XML Dialplan has the ability to test a number of conditions based upon variables with expressions; however, it needs to be understood that some variables may not be available for conditional testing until the first transfer or execute_extension is performed (see workarounds below).

Why

In essence the XML Dialplan is to be used for Call Routing rather than for complex or extensive conditional tests and evaluations. This is why FreeSWITCH makes Lua, JavaScript, Perl, Python and other APIs available since they are far better alternatives than coming up with a convoluted XML solution, or worse yet some arcane and convoluted acronym such as "AEL".

This may be confusing to former Asterisk users since the info application such as <action application="info"/> will in fact display the variables as if they are available for a conditional test when in fact they may not.

The reason for this is that FreeSWITCH does the **hunting** and the **executing** in two separate steps. First - based on conditions, actions and anti-actions - all applications that need to be executed are gathered. Second, that sequence of applications is executed. This means that channel variables set by the executed applications won't be available to conditions at hunting time.

This is why you may find that your XML condition is failing even though the variable and its value are displayed with the <action application="info"/>.

Workarounds

The workaround for this is to either implement the vast majority of your dialplan logic within Lua, JavaScript or one of the other Dialplan scripting languages, OR execute an extension which will make those variables you seek to do conditional evaluations on available for parsing within your XML Dialplan condition.

NOTE: Since svn rev 14906 it is possible for certain applications to be run **inline**. This means that they are executed at hunting time which has the effect that channel variables set by these applications **are** available to the following conditions at hunting time.

Actions and Anti-Actions

So far, we've seen example dialplan entries that contain conditions along with the actions that run when the conditions match.

You can also specify 'anti-actions' that run if the conditions for the extension 'are not met'.

In this example, the value of ${my_var} is compared with 12, and a message is logged for either result.

```
<condition field="${cond(${my_var} > 12 ? YES : NO)}" expression="^YES$">
  <action application="log" data="INFO ${my_var} is indeed greater than 12"/>
  <anti-action application="log" data="INFO ${my_var} is not greater than 12"/>
</condition>
```

Available Actions

See API Reference and Dialplan Functions

Inline Actions

You may set an extra attribute **inline="true"** on an action so that it will be executed during the hunting phase of the dialplan:

```
<action inline="true" application="set" data="some_var=some_val"/>
```

This makes it possible to have a condition in the following extension, that matches on the **${some_var}** field.

Note that the only applications that may be run inline are the ones that quickly get- or set some variable(s) and that don't access or modify the state of the current session.

Applications that are allowed to be run **inline** are:

- check_acl
- eval
- event
- export
- log
- presence
- set
- set_global
- set_profile_var
- set_user
- unset
- verbose_events
- cidlookup
- curl
- easyroute
- enum
- lcr

- nibblebill
- odbc_query

Also keep in mind that inline executed applications don't show up in your call detail records like normally run applications do.

Complete Syntax

```
<condition field="[{field_name}|${variable_name}|${api_func(api_args ${var_name})}]" expression=
  <action application="app name" data="app arg"/>
  <anti-action application="app name" data="app arg"/>
</condition>
<!-- any number of condition tags may follow where the same rules apply -->
</extension>
```

Other Dialplan Pearls of Wisdom

The dialplan is parsed once when the call hits the dialplan parser in the ROUTING state. With one pass across the XML the result will be a complete list of instructions installed into the channel based on parsed <action> or <anti-action> tags.

Those accustomed to Asterisk may expect the call to follow the dialplan by executing the applications as it parses them allowing data obtained from one action to influence the next action. This is not the case with the exception being the ${api func(api arg ${var_name})} field type where a pluggable api call from a module may be executed as the parsing occurs. This is meant to be used to draw real-time information such as date and time or other quickly accessible information and should **not** be abused.

Auto Hunt

You many turn on auto_hunt and then if the Extension name precisely equals the dialed number, FreeSWITCH will jump to this extension to begin the searching. It may or may not match the conditions, though.

Dialing through gateways

"gateway" is treated as a keyword by mod_sofia, it obviously means the call will be placed through a configured gateway. This is an exception for the pattern sofia/profilename/extension@ip-address.

If a gateway, for instance, is named "gw", the bridge string for sending a call to gw's extension 100 would be:

```
<extension name="testing">
  <condition field="destination_number" expression="^(100)$">
    <action application="bridge" data="sofia/gateway/gw/$1"/>
  </condition>
</extension>
```

"destination_number" is a FreeSWITCH variable; it shouldn't be changed.

Examples

NOTE: if you plan to include your extension in a separated .XML file:

- please disable or change enum extension if you don't need it
- add the tag <include> and close it with </include>

Example 1: Matching a condition

In the example below, the particular extension will be selected only if the IP address of the calling endpoint is 192.168.1.1. In the second condition, the dialed number is extracted in variable $1 and put in the data of the bridge application, in order to dial out to IP address 192.168.2.2

```
<extension name="Test1">
  <condition field="network_addr" expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^(\d+)$">
    <action application="bridge" data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

Note that the first condition field is TERMINATED by a / !!! The last condition field which contains the action/anti-action is terminated by a regular </condition> tag. Also note that the above example is NOT the same as below:

```
<extension name="Test1Wrong">
  <condition field="destination_number" expression="^(\d+)$"/>
  <condition field="network_addr" expression="^192\.168\.1\.1$">
    <action application="bridge" data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

The Test1Wrong example will not route the call properly, because the variable $1 will not have any value, since the destination number was matched in a different condition field.

You can also solve the Test1Wrong example by setting a variable in the first condition which you then use inside the second condition's action:

```
<extension name="Test1_2">
  <condition field="destination_number" expression="^(\d+)$">
    <action application="set" data="dialed_number=$1"/>
```

```
  </condition>
  <condition field="network_addr" expression="^192\.168\.1\.1$">
    <action application="bridge" data="sofia/profilename/${dialed_number}@192.168.2.2"/>
  </condition>
</extension>
```

Note that you cannot use a variable set inside an extension for further conditions/matches as the extension is evaluated when the action is called.

If you need to do different actions based on a variable set inside an extension you need to either use execute_extension or a transfer for the variable to be set.

Example 2: Matching multiple conditions (AND)

In this example we need to match a called number beginning with the prefix 1 AND match the incoming IP address at the same time.

```
<extension name="Test2">
  <condition field="network_addr" expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^1(\d+)$">
    <action application="bridge" data="sofia/profilename/$0@192.168.2.2"/>
  </condition>
</extension>
```

Here, although we match with the rule 1(\d+)$ we don't use the variable $1 which would contain only the rest of the dialed number with the leading 1 stripped off, we use the variable $0 which contains the original destination number.

Example 3: Stripping leading digits

In this example we need to match a called number beginning with 00 but we also need to strip the leading digits. Assuming that FreeSWITCH™ receives the number 00123456789 and we need to strip the leading 00 digits, then we can use the following extension:

```
<extension name="Test3.1">
  <condition field="destination_number" expression="^00(\d+)$">
    <action application="bridge" data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

On the other hand, if you anticipate receiving non-digits, or you want to match on more than just digits, use ".+" instead of "\d+" because \d+ matches numeric digits only, whereas a .+ will match all characters from the current position to the end of the string:

```
<extension name="Test3.2">
  <condition field="destination_number" expression="^00(.+)$">
    <action application="bridge" data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

NOTE: Please be careful with regular expressions containing (.*) or (.+). It's a good practice not to use catchall expressions because you cannot trust information from the sender. In nearly all dialplans, the capture group should be numeric so use \d.

Example 4: Adding a prefix

In this example we need to strip the leading digits as above, but we also need to place a new prefix before the called number. Assuming that FreeSWITCH™ receives the number 00123456789 and we need to replace the 00 with the 011, we can use the following extension:

```
<extension name="Test4">
  <condition field="destination_number" expression="^00(\d+)$">
    <action application="bridge" data="sofia/profilename/011$1@x.x.x.x"/>
  </condition>
</extension>
```

Example 5: SIP Profiles (dialing with different configurations)

In this example we will demonstrate the use of profiles when using a FreeSWITCH endpoint that supports profiles, like mod_sofia. Assuming that we want to use different call settings (codecs, DTMF modes, etc) for sending the calls to different IP addresses, we can create different profiles. For example, in the configuration of sofia.conf, we see an example profile named "test", which we rename to profile1 for this example, and add a profile2 for comparison:

```
<profile name="profile1">
  <param name="debug" value="1"/>
  <param name="rfc2833-pt" value="101"/>
  <param name="sip-port" value="5060"/>
  <param name="dialplan" value="XML"/>
  <param name="dtmf-duration" value="100"/>
  <param name="codec-prefs" value="PCMU@20i"/>
  <param name="codec-ms" value="20"/>
  <param name="use-rtp-timer" value="true"/>
</profile>
<profile name="profile2">
  <param name="debug" value="1"/>
  <param name="rfc2833-pt" value="101"/>
  <param name="sip-port" value="5070"/>
  <param name="dialplan" value="XML"/>
  <param name="dtmf-duration" value="100"/>
  <param name="codec-prefs" value="PCMA@20i"/>
  <param name="codec-ms" value="20"/>
  <param name="use-rtp-timer" value="true"/>
</profile>
```

The difference between the two profiles are in the codecs. The first uses G.711 u-law and the second G.711 A-law.

Continuing the examples above, we have:

```
<extension name="Test5ulaw">
  <condition field="network_addr" expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^1(\d+)$">
    <action application="bridge" data="sofia/profile1/$0@192.168.2.2"/>
```

```
  </condition>
</extension>
```

to send the call in G.711 uLaw and

```
<extension name="Test5alaw">
  <condition field="network_addr" expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^1(\d+)$">
    <action application="bridge" data="sofia/profile2/$0@192.168.2.2"/>
  </condition>
</extension>
```

Example 6: Calling registered user

This example shows how to bridge to devices that have registered with your FreeSWITCH box. In this example we assume that you have setup a sofia profile called 'local_profile' and your phones are registering with the domain example.com. Note the '%' instead of '@' in the data string.

```
<extension name="internal">
  <condition field="source" expression="mod_sofia" />
  <condition field="destination_number" expression="^(4\d+)">
    <action application="bridge" data="sofia/local_profile/$0%example.com" />
  </condition>
</extension>
```

Example 7: Action failover on failed action

The following example shows how it is possible to call another action if the first action fails.

If the first action is successful the call is bridged to 1111@example1.company.com and will exist until one of the parties hangs up. After this, no other processing will be done because the caller's channel is closed. (i.e.: 1111@example2.company.com is **not** called)

If the initial call to 1111@example1.company.com was **not** successful the channel will not be closed and the second action will be called.

```
<extension name="internal">
  <condition field="destination_number" expression="^1111">
    <action application="set" data="hangup_after_bridge=true"/>
    <action application="bridge" data="sofia/local_profile/1111@example1.company.com" />
    <action application="bridge" data="sofia/local_profile/1111@example2.company.com" />
  </condition>
</extension>
```

Note: If you have more than one action and the application of the first action

- DOES hangup the channel, the second action will NOT be called.
- DOES NOT hangup the channel, the second action will be called.

Example 8: Check user is authenticated

The following example requires that a caller be authenticated before passing through. It was yanked from a mailing list post.

```
<extension name="9191">
  <condition field="destination_number" expression="^9191$"/>
  <condition field="${sip_authorized}" expression="true">
    <anti-action application="reject" data="407"/>
  </condition>

  <condition>
    <action application="playback" data="/tmp/itworked.wav"/>
  </condition>
</extension>
```

Example 9: Routing DID to an extension

To route incoming calls which come in to a certain DID to a fixed extension 1001, do something LIKE the following (from a mailing list post) (where XXXxxxxxxx is the phone number of your incoming DID)

In public.xml:

```
  <extension name="test_did">
    <condition field="destination_number" expression="^(XXXxxxxxxx)$">
      <action application="transfer" data="$1 XML default"/>
    </condition>
  </extension>
```

and then in default.xml have something like this in the default context:

```
    <extension name="Local_Extension">
    <condition field="destination_number" expression="^(XXXxxxxxxx)$">
      <action application="set" data="dialed_ext=$1"/>
    </condition>
    <condition field="destination_number" expression="^${caller_id_number}$">
      <action application="set" data="voicemail_authorized=${sip_authorized}"/>
      <action application="answer"/>
      <action application="sleep" data="1000"/>
      <action application="voicemail" data="check default $${domain} ${dialed_ext}"/>
      <anti-action application="ring_ready"/>
      <anti-action application="set" data="call_timeout=10"/>
      <anti-action application="set" data="hangup_after_bridge=true"/>
      <anti-action application="set" data="continue_on_fail=true"/>
      <anti-action application="bridge" data="USER/1001@$${domain}"/>
      <anti-action application="answer"/>
      <anti-action application="sleep" data="1000"/>
      <anti-action application="voicemail" data="default $${domain} ${dialed_ext}"/>
    </condition>
  </extension>
```

(the 1001 in the "bridge" line is the extension we're ringing)

FYI, calls from the "public" go into the public context where they then need to be transferred to another more friendly context for processing, like default. That is why you add the entry to public and the 'data="$1 XML

PFC"' says to transfer called number $1 to the context PFC using XML dialplan. In the default context is where you actually ring the phone.

$${domain} a variable that it automatically fills in for you with your domain (most likely your IP) and the calling number. Just leave them as is.

Example 10: Route to a gateway extension with custom caller id

In this example we demonstrate an outgoing call with 10 digits from extension 1000 then route it to the asterlink.com gateway. This examples shows how to route for a specific extension and allows custom caller id for that extension.

```
<extension name="asterlink.com">
  <condition field="caller_id_number" expression="^1000$"/>
  <condition field="destination_number" expression="^(\d{10})$">
      <action application="set" data="effective_caller_id_number=8001231234"/>
      <action application="set" data="effective_caller_id_name=800 Number"/>
      <action application="bridge" data="sofia/gateway/asterlink.com/1208$1"/>
  </condition>
</extension>
```

Example 11: Route based on number prefix

In this example we demonstrate routing to different destination based on NPANXX. Also how to respond to the calling party with a different failure message than the destination sends to FreeSWITCH.

```
<extension>
  <condition field="network_addr" expression="^(66\.123\.321\.231|70\.221\.221\.221)$" break="on-
  <condition field="destination_number" expression="^\d+$" break="never">
  <action application="set" data="continue_on_fail=NORMAL_TEMPORARY_FAILURE,TIMEOUT,NO_ROUTE_DEST
  <action application="set" data="bypass_media=true"/>
  <action application="set" data="accountcode=myaccount"/>
  </condition>
  <condition field="destination_number" expression="^(1813\d+|1863\d+|1727\d+|1941\d+|404\d+)$" b
  <action application="bridge" data="sofia/outbound_profile/${sip_to_user}@switch1.mydomain.com"/
  <action application="info"/>
  <action application="respond" data="503"/>
  <action application="hangup"/>
  </condition>
  <condition field="destination_number" expression="^(1404\d+|1678\d+|1770\d+)$">
  <action application="bridge" data="sofia/outbound_profile/${sip_to_user}@switch2.mydomain.com"/
  <action application="info"/>
  <action application="respond" data="503"/>
  <action application="hangup"/>
  <anti-action application="respond" data="503"/>
  <anti-action application="hangup"/>
  </condition>
</extension>
```

Example 12: Handle calls which match no extension

In this example we demonstrate how to catch invalid extensions/Destinations.

- You need to add this extension at bottom of your dialplan before ENUM can get included.
- See mod_enum.

```
<extension name="catchall">
  <condition field="destination_number" expression=".*" continue="on-true">
   <action application="playback" data="misc/invalid_extension.wav"/>
  </condition>
</extension>
```

Example 13: Call Screening

In this example, we ask the caller for a name, connect to the called party and announce that name. The called party may then press 1 to accept the call, or hang up. If the called party hangs up, the caller is connected with voicemail.

```
 <extension name="screen">
   <condition field="destination_number" expression="^(\d{4})$">
     <action application="set" data="call_screen_filename=/tmp/${caller_id_number}-name.wav"/>
     <action application="set" data="hangup_after_bridge=true" />
     <action application="answer"/>
     <action application="sleep" data="1000"/>
     <action application="phrase" data="voicemail_record_name"/>
     <action application="playback" data="tone_stream://%(500, 0, 640)"/>
     <action application="set" data="playback_terminators=#*0123456789"/>
     <action application="record" data="${call_screen_filename} 7 200 2"/>
     <action application="set" data="group_confirm_key=1"/>
     <action application="set" data="fail_on_single_reject=true"/>
     <action application="set" data="group_confirm_file=phrase:screen_confirm:${call_screen_filen
     <action application="set" data="continue_on_fail=true"/>
     <action application="bridge" data="user/$1"/>
     <action application="voicemail" data="default $${domain} $1"/>
     <action application="hangup"/>
   </condition>
 </extension>
```

Example 14: Media recording

This extension is used to play/record media in audio (wav) format recording / playback extension

- Thanks to rupa for the help.

```
<extension name="recording">
  <condition field="destination_number" expression="^(2020)$">
    <action application="answer"/>
    <action application="set" data="playback_terminators=#"/>
    <action application="record" data="/tmp/recorded.wav 20 200"/>
  </condition>
</extension>

<extension name="playback">
  <condition field="destination_number" expression="^(2021)$">
    <action application="answer"/>
    <action application="set" data="playback_terminators=#"/>
    <action application="playback" data="/tmp/recorded.wav"/>
  </condition>
```

```
    </extension>
```

Example 15: Speaking Clock

This example will speak time using the Flite text to speech engine.

- See: mod_flite

```
<include>
  <extension name="SpeakTime">
    <condition field="destination_number" expression="^2910$">
      <action application="set" data="actime=${strftime(%H:%M)}"/>
      <action application="set" data="tts_engine=flite"/>
      <action application="set" data="tts_voice=slt"/>
      <action application="speak" data="It is +${actime}"/>
    </condition>
  </extension>
</include>
```

Example 16: Block certain codes

This extension example is to demonstrate how to block certain NPAs that you do not want to terminate based on caller id area codes and respond with SIP:503 to your origination so that they can route advance if they have other carrier to terminate to.

```
<extension name="blocked_cid_npa">
 <condition field="caller_id_number" expression="^(\+1|1)?((876|809)\d{7})$">
   <action application="respond" data="503"/>
   <action application="hangup"/>
 </condition>
</extension>
```

Example 17: Receive fax from inbound did

To use the predefined fax_receive extension in freeswitch/conf/dialplan/default.xml for inbound calls, put this in freeswitch/conf/dialplan/public/fax.xml:

```
<include>
  <extension name="incoming-fax">
    <condition field="destination_number" expression="^$${local_fax_number}$">
      <action application="set" data="domain_name=$${domain}"/>
      <action application="transfer" data="9178 XML default"/>
    </condition>
  </extension>
</include>
```

Then in freeswitch/conf/vars.xml you set your fax number to 1234 or whatever:

```
<X-PRE-PROCESS cmd="set" data="local_fax_number=1234"/>
```

Example 18: Add international call prefix to effective_caller_id_number on incoming BRI calls

When using FreeTDM with zaphfc on a BRI line, the incoming calls received will not contain the international call prefix in the caller_id_number. This extension adds it to the effective_caller_id_number.

The international call prefix is explained here: http://en.wikipedia.org/wiki/International_prefix.

In Germany, the international call prefix is called "Verkehrsausscheidungsziffer" (VAZ), see http://de.wikipedia.org/wiki/Verkehrsausscheidungsziffer.

The international call prefix is never transmitted. It can be predicted by looking at the ToN information.

```
<extension name="Add-VAZ" continue="true">
<!--
On incoming BRI calls, the Verkehrsausscheidungsziffer (VAZ) is dropped.
This extension adds it again to the caller_id_number.
TODO: add support for international numbers
-->
        <condition field="source" expression="^mod_freetdm$">
        <action application="set" data="effective_caller_id_number=0${caller_id_number}"/>
        </condition>
</extension>
```

Example 19: DISA

Be able to dial into FS box and get a dialtone to dial again, just like in Asterisk's DISA()

In FS/conf/dialplan/public/*.xml

```
<!-- -->
<!-- -->
<!-- -->
<!-- CAUTION!!! We TRANSFER here. Possible security breach. CAUTION!!! -->
<!-- -->
<!-- -->
<!-- -->
<extension name="incoming-bri-wor">
        <condition field="destination_number" expression="^(disa_target)$">
                <action application="answer"/>
                <action application="start_dtmf"/>
                <action application="play_and_get_digits" data="2 5 3 37000 #
                        $${base_dir}/sounds/en/us/callie/ivr/8000/ivr-please_enter_pin_followed_b
                        $${base_dir}/sounds/en/us/callie/ivr/8000/ivr-pin_or_extension_is-invalid
                        digits ^$${DISA_PASSWORD}$"/>
                <action application="transfer" data="$1 XML default"/>
        </condition>
</extension>
```

In FS/conf/dialplan/default/03_DISA.xml

```
<!--
        DISA - allow to dial into the box and get a dialtone like new trunk
-->
```

```
<include>
<extension name="DISA for FS">
        <condition field="destination_number" expression="^(disa_target)$">
                <action application="answer"/>
                <action application="read" data="2 15 'tone_stream://%(10000,0,350,440)' digits 3
                <action application="execute_extension" data="${digits}"/>
                <action application="transfer" data="disa_target XML default"/>
        </condition>
</extension>
</include>
```

Please replace "disa_target" to you extension number.

You can use this technique to dial into your box from a pstn / mobile phone and get a dialtone to do anything with.

Example 20: Fix invalid caller ID

If you run into a problem where your B-Leg do not like the invalid caller ID; for instance, you have an INVITE message with From: header as From: <sip:Unavailable@Unavailable.invalid:5060> You can force the invalid number to be replaced by a fixed caller ID number. The following example checks for a valid NANPA CLID:

```
<extension name="invalid_caller_id_fix" continue="true">
  <condition field="caller_id_number" expression="^1?([2-9]\d{2}[2-9]\d{6})$">
    <action application="set" data="effective_caller_id_number=$1"/>
    <anti-action application="set" data="effective_caller_id_number=2135551212"/>
  </condition>
</extension>
```

Example 21: Block outbound caller ID

To have caller ID block for calling party by dialing *67 follows by the dial number, you can do the following:

```
<extension name="block_caller_id">
  <condition field="destination_number" expression="^\*67(\d+)$">
    <action application="privacy" data="full"/>
    <action application="set" data="sip_h_Privacy=id"/>
    <action application="set" data="privacy=yes"/>
    <action application="transfer" data="$1 XML default"/>
  </condition>
</extension>
```

Example 22: Play MOH while doing a database lookup

If you want to play MOH while doing a data dip that takes a long time, the non-ESL way to do this by making the dialplan use the FSAPI via variable expansion to call luarun on the script. This way a new thread will be launched to execute the Lua script.

```
<extension name="Get_Data">
  <condition field="destination_number" expression="^(Get_Data)$">
        <action application="play_and_get_digits" data="4 16 3 7000 # phrase:Enter_Case_Numbe
```

```
            <action application="set" data="x=${expand(luarun GetDataFromLDAP.lua ${case_number}
            <action application = "playback" data="/tmp/LongMusicFile.wav"/>
        </condition>
    </extension>
```

In the Lua script you can use the "uuid" argument passed, to break the MOH or transfer to another extension

```
--GetDataFromLDAP.lua
key = argv[1]
sessionId = argv[2]
api = freeswitch.API()

--Do your data dips here

--Once the database operations are done you can simply stop the MOH or transfer to another extens
api:execute("uuid_break", sessionId) -- break the MOH
```

SIP-Specific Dialstrings

SIP dialing has several options. Here are some aspects of what you might call the anatomy of a SIP dialstring.

Dialing A SIP URI

Basic syntax is: sofia/my_profile/user@host Host can be a name or an IP address, for example:

```
sofia/my_profile/1234@192.168.0.1
```

This would dial 1234 at host 192.168.0.1 via the profile "my_profile". If you use a name instead of an IP address, Sofia will try to resolve the name as a NAPTR or SRV record before trying it as a standard A record.

Dialing A Registered User

There are two options depending upon whether or not there is an alias for the domain.
Without an alias you can do this:

```
sofia/my_profile/1234%mydomain.com
```

If you have an alias for the domain then this syntax is valid:

```
sofia/mydomain.com/1234
```

Note how the profile does not need to be explicitly supplied in the dialstring.

Also you can do it this way for users defined in the directory:

```
user/1234@mydomain.com
```

Dialing Through A Gateway (SIP Provider)

A gateway is a means for making outbound calls through a SIP provider. For example:

```
sofia/gateway/mygateway.com/1234
```

This will dial through the gateway named "mygateway.com" to user 1234.
Note how there is no need to append anything after the user "1234"
This is an example of how not to do it:

```
sofia/gateway/mygateway.com/1234@mygateway.com  <==== WRONG WRONG WRONG
```

Dialing With A Specific Transport

Sometimes you will need to specify the transport, for example TCP, UDP, TLS, or SCTP.
This can be done by appending a semicolon and the transport method. For example:

```
sofia/my_profile/1234@192.168.0.1;transport=tcp
```

Specifying The Codec

Occasionally you may want to force the system to use a specific codec. This syntax will accomplish that:

```
{absolute_codec_string=XXXX}sofia/my_profile/user@your.domain.com
```

In this example, XXXX represents the codec to be used. The possible codec values are listed here.
Additional dialstring examples from Absolute Codec String variable.

Getting Fancy With PortAudio

If you have PortAudio running and would like to specify the codec you need to originate first and bridge second:

```
originate {absolute_codec_string=XXXX}sofia/default/foo@bar.com bridge:portaudio/auto_answer inli
```

Changing the SIP Contact user

FreeSWITCH normally uses mod_sofia@ip:port for the internal SIP contact. To change this to foo@ip:port, there is a variable, sip_contact_user:

```
{sip_contact_user=foo}sofia/my_profile/1234@192.168.0.1;transport=tcp
```

Using a Custom SIP URI

FreeSWITCH allows you to specify custom URI's as needed. For example, you may need to interoperate with equipment that accepts a URI only if it is formatted in a particular way. The key is to prefix your SIP URI with "sip:" in the dialstring. For example:

```
sofia/my_profile/sip:xxxx;phone-context=cdp.udp@somedomain.com;user=phone
```

The above example will send the the URI exactly as specified after the "sip:" prefix.

Testing the dialplan with a command line

```
originate loopback/<destination number>/<mycontext> hangup inline
```

- Note: You can also set your variables to match your dialplan requirement. See below example:

```
originate {toll_allow=international}loopback/0116628888888/default hangup inline
```

Setting up SIP Diversion Header for call forward

```
 <action application="export" data="sip_h_Diversion=<sip:2134445555@1.2.3.4>;reason=unavailable"/
```

Related

- Dialplan
- Dialplan ARRAYS
- Freeswitch IVR Originate
- Channel Variables
- Dialplan Recipes
- Misc._Dialplan_Tools_bridge