FreeSWITCH 1.6 Cookbook

FreeSWITCH is an open source carrier-grade telephony platform designed to facilitate the creation of voice, chat, and video applications, via phones and web browsers. It is scalable, carrier-ready, and easy-to-program for converged communication and VoIP. The technology serves SIP, WebRTC, PSTN, FAX, PBX, VERTO, and all the relevant channels essential to stay connected in today’s world.

In the FreeSWITCH 1.6 Cookbook, members of the FreeSWITCH development team share some of their hard-earned knowledge with you. Use this knowledge to improve and expand your FreeSWITCH installations.

What this book will do for you...

- Configure users and phones as well as connections to VoIP providers
- Control FreeSWITCH remotely with the powerful event socket interface
- Route inbound and outbound calls
- Insert CDRs into a database
- Enable text-to-speech in your voice applications
- Set up SSL certificates and release services to WebRTC
- Write JavaScript WebRTC clients for real-time data/video/audio in browsers
- Grasp the FreeSWITCH security best practices and Lua application programming knowledge

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Over 45 practical recipes to empower you with the latest FreeSWITCH 1.6 features
In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Routing Calls'
- A synopsis of the book’s content
- More information on FreeSWITCH 1.6 Cookbook
About the Authors

Anthony Minessale II is the primary author and founding member of the FreeSWITCH open source softswitch. He has spent almost 20 years working with open source software. In 2001, he spent a great deal of time as an Asterisk developer and authored numerous features and fixes to that project. Anthony started coding a new idea for an open source voice application in 2005. The FreeSWITCH project was officially opened to the public on January 1, 2006. In the years that followed, Anthony has actively maintained and led software development for this project.

Michael S Collins is a telephony and open source software enthusiast. Having worked as a PBX technician for 5 years and the head of IT for a call center for more than 9 years, he is a PBX veteran. He is an active member of the FreeSWITCH community and has coauthored FreeSWITCH Cookbook, by Packt Publishing in 2012. Michael lives in Central California with his wife and two children.
Giovanni Maruzzelli (available at OpenTelecom.IT) is heavily engaged with FreeSWITCH. In it, he wrote interfacing with Skype and cellular phones. He's a consultant in the telecommunication sector, developing software and conducting training courses for FreeSWITCH, SIP, WebRTC, Kamailio, and OpenSIPS.

An Internet technology pioneer, he was the cofounder of Italia Online in 1996. It is the most popular Italian portal and consumer ISP. Also, he was the architect of its Internet technologies (www.italiaonline.it). Then, Giovanni was the supervisor of Internet operations and the architect of the first engine for paid access to ilsole24ore.com, the most read financial newspaper in Italy, and its databases (migrated from the mainframe).

After that, he was the CEO of the venture-capital-funded company Matrice, developing telemail unified messaging and multiple-language phone access to e-mail (text to speech). He was also the CTO of the incubator-funded company Open4, an open source managed applications provider.

For 2 years, Giovanni worked in Serbia as an Internet and telecommunication investment expert for IFC, an arm of The World Bank.

Since 2005, he has been based in Italy and serves ICT and telecommunication companies worldwide.
FreeSWITCH is increasingly becoming the "serious choice" for companies to base their products and offerings on. Its usage is widespread, scaling from Raspberry Pis to "Big Irons" in the data center.

There is a growing need for books and training, and with Packt Publishing, we decided to begin serving this burgeoning demand. This cookbook is a primer; then there will be a *Mastering FreeSWITCH* book, followed by a new edition of the *classic* FreeSWITCH book.

Obviously, nothing can beat a training camp or codeveloping in collaboration with an old hand, but these FreeSWITCH titles will form the basis on which a company or a consultant can begin embracing, deploying, and implementing FreeSWITCH.

This book is a complete update, rewrite, and integration of the old FreeSWITCH cookbook. This new edition covers FreeSWITCH 1.6.x, and a lot of new ground.

All the examples here have been updated and tested with the new FreeSWITCH series, while a new section has been added about connecting to Skype, and two entire chapters are on WebRTC and Lua programming.

Anthony Minessale II, Giovanni Maruzzelli

July 5 2015

**What this book covers**

*Chapter 1, Routing Calls*, shows that getting calls from one endpoint to another is the primary function of FreeSWITCH. This chapter discusses techniques of efficiently routing calls between phones and service providers.

*Chapter 2, Connecting Telephones and Service Providers*, assists in quickly getting your FreeSWITCH server connected to other VoIP devices. Telephones and service providers have specific requirements for connecting to FreeSWITCH.
Chapter 3, *Processing Call Detail Records*, discusses a number of ways to extract CDR data from your FreeSWITCH server. Call detail records, or CDRs, are very important for businesses.

Chapter 4, *External Control*, presents a number of real-world examples of controlling FreeSWITCH from an external process. FreeSWITCH can be controlled externally by the powerful and versatile event socket interface.

Chapter 5, *PBX Functionality*, is the largest chapter in this book. This chapter shows how to deploy features such as voicemail, conference calls, faxing, IVRs, and more, which most telephone systems have, in a FreeSWITCH server.

Chapter 6, *WebRTC and Mod_Verto*, features the new disruptive technology that allows real-time audio/video/data-secure communication from hundreds of millions of browsers. FreeSWITCH is ready to serve as a gateway and an application server.

Chapter 7, *Dialplan Scripting with Lua*, covers Lua, the scripting language of choice for programming complex logic in FreeSWITCH. Accessing databases, calling web servers, and interacting with user's choices now becomes easy.
In this chapter, we will discuss routing calls in various scenarios, as follows:

- Internal calls
- Incoming DID (also known as DDI) calls
- Outgoing calls
- Ringing multiple endpoints simultaneously
- Ringing multiple endpoints sequentially (simple failover)
- Advanced multiple endpoint calling with enterprise originate
- Time-of-day routing
- Manipulating SIP To: headers on registered endpoints to reflect DID numbers

**Introduction**

Routing calls is at the core of any FreeSWITCH server. There are many techniques for accomplishing the surprisingly complex task of connecting one phone to another. However, it is important to make sure that you have the basic tools necessary to complete this task.

The most basic component of routing calls is the **dialplan**, which is essentially a list of actions to perform depending upon which digits were dialed (as we will see in some of the recipes in this book, there are other factors that can affect routing of calls). The dialplan is broken down into one or more **contexts**. Each context is a group of one or more **extensions**. Finally, each extension contains specific **actions** to be performed on the call. The dialplan processor uses **regular expressions**, which are a pattern-matching system used to determine which extensions and actions to execute.

To make best use of the recipes in this chapter, it is especially important to understand how to use regular expressions and the three contexts in the default configuration.
FreeSWITCH uses Perl-compatible regular expressions (PCRE) for pattern matching. Consider this dialplan excerpt:

```xml
<extension name="example">
  <condition field="destination_number" expression="^(10\d\d)$">
    <action application="log" data="INFO dialed number is \[$1\]"/>
  </condition>
</extension>
```

This example demonstrates the most common uses of regular expressions in the dialplan: matching against the destination_number field (that is, the digits that the user dialed) and capturing, using parentheses, the matched value in a special variable named $1. Let's say that a user dials 1025. Our example extension will match 1025 against the ^10\d\d$ pattern and determine that this is indeed a match. All actions inside the condition tag will be executed. The action tag in our example will execute the log application. The log application will then print a message to the console, using the INFO log level, which will be in green text by default. The value in $1$ is expanded (or interpolated) when printed:

```
2015-02-22 15:15:50.664585 [INFO] mod_dptools.c:1628 dialed number is [1025]
```

Understanding these basic principles will help you create effective dialplan extensions.

For more tips on using regular expressions, be sure to visit http://freeswitch.org/confluence/display/FREESWITCH/Regular+Expression.

### Important dialplan contexts in the default configuration

Contexts are logical groups of extensions. The default FreeSWITCH configuration contains three contexts:

- default
- public
- features

Each of these contexts serves a purpose, and knowing about them will help you leverage their value for your needs.
The default context

The most used context in the default configuration is the default context. All users whose calls are authenticated by FreeSWITCH will have their calls passing through this context, unless there have been modifications. Some common modifications include using ACLs or disabling authentication altogether (see the The public context section that follows). The default context can be thought of as internal in nature; that is, it services users who are connected directly to the FreeSWITCH server, as opposed to outside callers (again, see the The public context section).

Many characteristics related to PBX (Private Branch Exchange) are defined in the default context, as are various utility extensions. It is good to open conf/dialplan/default.xml and study the extensions there. Start with simple extensions such as show_info, which performs a simple info dump to the console, and vmain, which allows a user to log in to their voicemail box.

A particularly useful extension to review is Local_Extension. This extension does many things, as follows:

- Routes calls between internal users
- Sends calls to the destination user's voicemail on a no-answer condition
- Enables several in-call features with bind_meta_app
- Updates (via "hash" commands) the local calls database to allow call return and call pickup

Many of the techniques employed in Local_Extension are discussed in this chapter (see the The features context section for a discussion on the in-call features found in this extension).

The public context

The public context is used to route incoming calls that originate from outside the local network. Calls that initially come to the public context are treated as untrusted. If they are not specifically routed to an extension in the default context, then they are simply disconnected. As mentioned before, disabling authentication or using ACLs to let calls into the system will route them into the public context (this is a security precaution, which can be overridden if absolutely required). We will use the public context in the Incoming DID (also known as DDI) calls recipe.

The features context

The features context is used to make certain features available for calls that are in progress. Consider this excerpt from Local_Extension in conf/dialplan/default.xml:

```xml
<action application="bind_meta_app" data="1 b s execute_extension::dx XML features"/>
```
Routing Calls

This is just one of several features that are enabled for the recipient of the call. The bind_meta_app application listens on the audio stream for a touch-tone * followed by a single digit. The preceding example is a blind transfer. If the called user dials *1, then the execute_extension::dx XML features command is executed. In plain words, this command says, “Go to the features context of the XML dialplan and execute the extension whose destination number is dx.” In conf/dialplan/features.xml, there is the following extension:

```xml
<extension name="dx">
  <condition field="destination_number" expression="^dx$">
    ...
  </condition>
</extension>
```

The dx extension accepts some digits from the user and then transfers the caller to the destination that the user keyed in.

This process demonstrates several key points:

- Calls can be transferred from one dialplan context to another
- The features context logically isolates several extensions that supply in-call features
- The bind_meta_app dialplan application is one of the means of allowing in-call features

Understanding that calls can flow from one context to another even after they are in progress is an important concept to grasp when addressing your call routing scenarios.

**Internal calls**

Calling local extensions is very simple once you know what needs to happen. In this case, we will see how to add a new user and make their phone available for calling.

**Getting ready**

If you are using the default configuration, then users 1000 through 1019 are preconfigured, both in the directory and the dialplan. To add a user beyond this range to the directory, it is generally easier to run the add_user script, found in the FreeSWITCH source directory under scripts/perl. For example, to add user 1020 to the directory, launch this script from the FreeSWITCH source directory, specifying the user ID on the command line:

```
scripts/perl/add_user 1020
```

You can also specify a range of users:

```
scripts/perl/add_user --users=1020-1029
```
You will see a note about the number of users added to the directory. If you have the Regexp::Assembly CPAN module installed, then the script will also generate a couple of sample regular expression patterns, which you can then use in the dialplan. For our example, we added a range of users from 1020 to 1029 to the directory, and then we'll add them to the dialplan.

**How to do it...**

1. Open the conf/dialplan/default.xml file in a text editor. Locate the Local_Extension entry:

   ```xml
   <extension name="Local_Extension">
   <condition field="destination_number"
   expression="^(10[01][0-9])$">
   
   2. Edit the expression in the `<condition>` tag to account for our new users. The `^(10[012][0-9])$` expression pattern will do what we need (look closely to see the difference). The new line will be as follows:

   ```xml
   <condition field="destination_number"
   expression="^(10[012][0-9])$">
   
   3. Save the file and then execute reloadxml from fs_cli.

**How it works...**

Local_Extension is the default dialplan entry that allows directory users to be called. Remember that *simply adding a user to the directory does not mean that the user can be dialed*. (However, it does usually mean that the user can make outbound calls.) So, in order for your new user to be reachable, you need to add their user ID to the dialplan. By default, Local_Extension has a regular expression that will match 1000, 1001, and so on up to 1019. After adding that number range, it is necessary to modify the regular expression to account for those new numbers. In our example, we added user IDs 1020 through 1029, so we need to match these. We use this regular expression:

```
^(10[012][0-9])$
```
Routing Calls

This matches 1000 through 1029. Let’s say we have added another block of user IDs with the range from 1030 to 1039. We can modify our regular expression to catch them as well:

^(10[0123][0-9])$

It is considered best practice not to add a large range of dialable numbers to Local Extension without having the corresponding users in the directory. Doing so can make troubleshooting dialplan issues more difficult.

As a reminder, be sure to execute the reloadxml command each time you modify the regular expression (the changes you make to your XML configuration files will not take effect until they are loaded into the memory, which is what the reloadxml command does).

See also

- The Creating users section in Chapter 5, PBX Functionality

Incoming DID (also known as DDI) calls

Phone calls coming in from the Public Switched Telephone Network (PSTN) are often called DID or DDI calls. DID stands for Direct Inward Dialing, while DDI means Direct Dial In; both acronyms refer to the same thing. DID numbers are delivered by your telephone service provider. They can be delivered over VoIP connections (such as a SIP trunk) or via traditional telephone circuits, such as PRI lines. These phone numbers are sometimes called DID numbers or external phone numbers.

Getting ready

Routing a call requires two pieces of information: the phone number being routed and a destination for that phone number. In our example, we will use a DID number 8005551212. Our destination will be user 1000. Replace these sample numbers with the appropriate values for your setup.

How to do it...

1. Create a new file in conf/dialplan/public/ named 01_DID.xml. Add this text to it:

```xml
<include>
<extension name="public_did">
<condition field="destination_number" expression="^(8005551212)$">
<action application="set" data="domain_name=${domain}"/>
<action application="transfer" data="1000 XML default"/>
```
2. Save the file and then execute `reloadxml` from `fs_cli`.

**How it works...**

All calls that come in to the FreeSWITCH server from outside (as well as internal calls that are not authenticated) are initially handled in the public context (dialplan contexts were discussed in more detail in this chapter's introduction) of the XML dialplan. Once the call hits the public context, we try to match the `destination_number` field. The `destination_number` is generally the DID number (see the *There's more...* section for some caveats). Once we match the incoming number, we set the `domain_name` channel variable to the default domain value, and then transfer the call to user 1000. (FreeSWITCH is domain-based in a way similar to e-mails. Most systems have only a single domain, though FreeSWITCH supports multiple domains.) The actual transfer happens with this dialplan entry:

```xml
<action application="transfer" data="1000 XML default"/>
```

In plain words, this tells FreeSWITCH to transfer the call to extension 1000 in the default context of the XML dialplan. The default context contains the `Local_Extension` that matches "1000" as `destination_number` and handles the calls to users' telephones.

**There's more...**

Keep in mind that the expression for `destination_number` must match what the provider sends to FreeSWITCH, not necessarily what the calling party actually dialed. There are providers that send DID information in various formats, such as these:

- `8005551212`
- `18005551212`
- `+18005551212`

The expression must match what the provider sends. One way to accomplish this is to have a few *optional characters in the pattern*. This pattern matches all three formats you just saw:

```xml
<condition field="destination_number"
expression="^\+?.?1?(8005551212)$">  
```

The `\+?` value means "optionally match the literal + character," and the `1?` value means "optionally match the literal digit 1." Now our pattern will match all of the three formats that are commonly used in North America. (Technically, our pattern will also match `+8005551212`, but we are not concerned about that. However, the pedantic admin might be, so they can use the `^\(\+1\)?1?(8005551212)$` pattern instead.)
Routing Calls

See also

- The Configuring a SIP gateway section in Chapter 2, Connecting Telephones and Service Providers

Outgoing calls

In order to make your system useful, you need a way to dial out to the "real world". This recipe will cover dialing out to the PSTN and allow you to connect to landlines, cellular phones, and so on. In this recipe, we'll make an extension that will allow an outbound call to any valid US number. We'll attempt to complete the call using the gateway named our_sip_provider (see the Configuring an SIP Gateway section in Chapter 2, Connecting Telephones and Service Providers).

Getting ready

Making outbound calls requires you to know the numbering format that your provider requires. For example, do they require all 11 digits for US dialing? Or will they accept 10? In our example, we're going to assume that our provider will accept a 10-digit format for US dialing (for example, without the international prefix "1").

How to do it...

Routing outbound calls is simply a matter of creating a dialplan entry:

1. Create a new file in conf/dialplan/default/ named outbound_calls.xml. Then add the following text:

```xml
<include>
  <extension name="outbound_calls">
    <condition field="destination_number" expression="^1?([2-9]\d{2}[2-9]\d{6})$">
      <action application="bridge" data="sofia/gateway/our_sip_provider/$1"/>
    </condition>
  </extension>
</include>
```

2. Save your XML file and issue the reloadxml command at fs_cli.
How it works...

Assuming you have a phone set up on the default context, your regular expression will match any destination_number that follows the US dialing format (10 or 11 digits), and send the call to our_sip_provider in a 10-digit format. The format in regexp is as follows: optional "1", then one digit between 2 and 9, then two digits, then one digit between 2 and 9, and finally six digits. Only the part after the optional digit "1" is captured by the parentheses and passed down in the $1 variable.

There's more...

The regular expression matching in FreeSWITCH allows the privilege of having very powerful conditions. You can also match caller_id_number to route calls from a user calling from extension 1011 out to the second gateway called our_second_sip_provider, while everyone else will be sent through our_sip_provider. Consider the following alternative outbound_calls.xml file:

```xml
<include>
  <extension name="outbound_calls_from_1011">
    <condition field="caller_id_number" expression="^1011$"/>
    <condition field="destination_number"
      expression="^1?([2-9]\d{2}[2-9]\d{6})$">
      <action application="bridge"
        data="sofia/gateway/our_second_sip_provider/$1"/>
    </condition>
  </extension>
  <extension name="outbound_calls">
    <condition field="destination_number"
      expression="^1?([2-9]\d{2}[2-9]\d{6})$">
      <action application="bridge"
        data="sofia/gateway/our_sip_provider/$1"/>
    </condition>
  </extension>
</include>
```

Note that we have two extensions. The first one tries to match the caller_id_number field to the value 1011. If it matches 1011, then the call gets sent to the our_second_sip_provider gateway. Otherwise, the next extension is matched and the call goes to the our_sip_provider gateway. Note that we use $1 to capture the matching value in the conditions' expressions. In each case, we capture exactly 10 digits, which correspond to the area code (three digits), exchange (three digits), and phone number (four digits). These are North American Numbering Plan (NANP) numbers. The regular expressions used to capture the format of dialed digits vary, depending upon the country.
Routing Calls

Regular expressions can be a challenge. There are a number of examples with explanations on the FreeSWITCH wiki. See http://freeswitch.org/confluence/display/FREESWITCH/Regular+Expression for further details.

See also

- The Configuring an SIP phone to register with FreeSWITCH and Configuring an SIP gateway sections in Chapter 2, Connecting Telephones and Service Providers

Ringing multiple endpoints simultaneously

FreeSWITCH makes it easy to ring multiple endpoints simultaneously within a single command.

Getting ready

Open conf/dialplan/default.xml in a text editor or create a new XML file in the conf/dialplan/default/ subdirectory.

How to do it...

Add a comma-separated list of endpoints to your bridge (or originate) application. For example, to ring userA@local.pbx.com and userB@local.pbx.com simultaneously, use an extension like this:

```xml
<extension name="ring_simultaneously">
  <condition field="destination_number" expression="^(2000)$">
    <action application="bridge"
      data="{ignore_early_media=true}sofia/internal/
        userA@local.pbx.com,sofia/internal/userB@local.pbx.com"/>
  </condition>
</extension>
```

How it works...

Putting comma-separated endpoints into the argument to bridge causes all the endpoints in that list to be dialed simultaneously. It sounds simple; however, there are several factors to consider when ringing multiple devices simultaneously in a real environment. The bridge application will connect the call to whoever sends the media first. This includes early media (ringing). To put this in other words, if you bridge a call to two parties and one party starts sending a ringing signal back to you, that will be considered media and the call will be connected to that party. Ringing of the other phones will cease.
If you notice that calls always go to a specific number on your list of endpoints versus ringing all numbers, or that all phones ring for a moment before ringing only a single number, it means that your call may be getting bridged prematurely because of early media. Notice that we added `ignore_early_media=true` at the beginning of the dial string. As its name implies, `ignore_early_media` tells the bridge application not to connect the calling party to the called party when receiving early media (such as a ringing or busy signal). Instead, `bridge` will only connect the calling party to the called party who actually answers the call. In most cases, it is useful to `ignore early media when ringing multiple endpoints` simultaneously.

**There's more...**

In some scenarios, you may also wish to ring specific devices for a limited amount of time. You can apply the `leg_timeout` parameter to each leg of the bridge to specify how long to ring each endpoint like this:

```
<action application="bridge"
  data="[leg_timeout=20]sofia/internal/userA@local.pbx.com,
     [leg_timeout=30]sofia/internal/userB@local.pbx.com"/>
```

In this example, the `userA` user's phone will ring for a maximum of 20 seconds, while the `userB` user's phone will ring for a maximum of 30 seconds.

**Call legs and the leg_timeout variable**

The `leg_timeout` variable is unique in the sense that it implies the ignoring of early media. When using the `leg_timeout` variable on each call leg in a bridge attempt, there is no need to explicitly use `{ignore_early_media=true}` in the bridge argument. For a more thorough discussion of using `{ and `} (curly brackets) versus `[ and ]` (square brackets), see [http://freeswitch.org/confluence/display/FREESWITCH/ChannelVariables#ChannelVariablesinDialstrings](http://freeswitch.org/confluence/display/FREESWITCH/ChannelVariables#ChannelVariablesinDialstrings).

This method of calling multiple parties works well for a small number of endpoints. However, it does not scale to dozens or more users. Consider using a FIFO queue in such an environment (FreeSWITCH's `mod_fifo` is discussed at length at [http://freeswitch.org/confluence/display/FREESWITCH/mod_fifo](http://freeswitch.org/confluence/display/FREESWITCH/mod_fifo)).

**See also**

- The *Ringing multiple endpoints sequentially (simple failover)* recipe that follows for an example of ringing a group of endpoints one at a time, which includes an expanded discussion of using call timeouts
Routing Calls

Ringing multiple endpoints sequentially (simple failover)

Sometimes it is necessary to ring additional endpoints, but only if the first endpoint fails to connect. The FreeSWITCH XML dialplan makes this very simple.

Getting ready

Open `conf/dialplan/default.xml` in a text editor or create a new XML file in the `conf/dialplan/default/` subdirectory.

How to do it...

Add a pipe-separated list of endpoints to your `bridge` (or `originate`) application. For example, to ring `userA@local.pbx.com` and `userB@local.pbx.com` sequentially, use an extension like this:

```xml
<extension name="ring_sequentially">
  <condition field="destination_number" expression="^\(2001\)$">
    <action application="bridge"
      data="\{ignore_early_media=true\}sofia/internal/
      userA@local.pbx.com|sofia/internal/userB@local.pbx.com"/>
  </condition>
</extension>
```

How it works...

Putting pipe-separated endpoints in the argument to `bridge` (or `originate`) causes all the endpoints in that list to be dialed sequentially. The first endpoint on the list that is successfully connected will be bridged, and the remaining endpoints will not be dialed. There are several factors to consider when ringing multiple devices sequentially.

Notice that we added `ignore_early_media=true` at the beginning of the dial string. As its name implies, `ignore_early_media` tells the `bridge` application not to connect the calling party to the called party when receiving early media (such as a ringing or busy signal). Instead, `bridge` will only connect the calling party if the called party actually answers the call. In most cases, you will need to ignore early media when dialing multiple endpoints.

There's more...

Handling different failure conditions can be a challenge. FreeSWITCH has a number of options that let you tailor `bridge` and `originate` to your specific requirements.
Handling busy and other failure conditions

For example, when calling a user who is on the phone, one service provider might return SIP message 486 (USER_BUSY), whereas many providers might simply send a SIP 183 with SDP and a media stream with a busy signal. In the latter, how will the bridge application know that there is a failure if it is ignoring the early media that contains the busy signal? FreeSWITCH gives us a tool that allows us to monitor early media even while "ignoring" it.

Consider two very common examples of failed calls where the failure condition is signaled in-band:

- Calling a line that is in use
- Calling a disconnected phone number

These conditions are commonly communicated to the caller via specific sounds: busy signals and special information tones, or SIT tones. In order for the early media to be meaningful, we need to be able to listen for specific tones or frequencies. Additionally, we need to be able to specify that certain frequencies mean different kinds of failure conditions (this becomes important for reporting, as in call detail records or CDRs). The tool that FreeSWITCH provides us with is a special channel variable called `monitor_early_media_fail`. Its use is best illustrated with an example:

```
<action application="bridge" data="{ignore_early_media=true,
    monitor_early_media_fail=user_busy:2:480+620!
    destination_out_of_order:2:1776.7}sofia/internal/
    userA@local.pbx.com|sofia/internal/userB@local.pbx.com"/>
```

Here, we have a bridge application that ignores early media and sets two failure conditions: one for busy and one for destination_out_of_order. We specify the name of the condition we are checking, the number of hits, and the frequencies to detect. The format for `monitor_early_media_fail` is as follows:

```
condition_name:number_of_hits:tone_detect_frequencies
```

The `user_busy` condition is defined as `user_busy:2:480+620`. This condition looks for both 480 Hz and 620 Hz frequencies (which is the USA busy signal), and if they are detected twice, then the call will fail. The exclamation mark (!) is the delimiter between the conditions. The `destination_out_of_order` condition is defined like this:

```
destination_out_of_order:2:1776.7.
```

This looks for two occurrences of 1776.7 Hz, which is a common SIT tone frequency in the USA (there is a nice introductory article on SIT tones at [http://en.wikipedia.org/wiki/Special_information_tones](http://en.wikipedia.org/wiki/Special_information_tones)). If 1776.7 Hz is heard twice, then the call will fail as destination out of order.

When using `monitor_early_media_fail`, only the designated frequencies are detected. All other tones and frequencies are ignored.
Routing Calls

Handling no-answer conditions

Handling a no-answer condition is different from busy and other in-band errors. In some cases, the service provider will send back SIP message 480 (NO_ANSWER), whereas others will send a ringing signal (SIP 183) in the early media until the caller decides to hang up. The former scenario is handled automatically by the bridge application. The latter can be customized with the use of special timeout variables:

- **call_timeout**: Sets the call timeout for all legs when using bridge
- **originate_timeout**: Sets the call timeout for all legs when using originate
- **leg_timeout**: Sets a different timeout value for each leg
- **originate_continue_on_timeout**: Specifies whether or not the entire bridge or originate operation should fail if a single call leg times out

By default, each call leg has a timeout of 60 seconds and bridge, or originate, will stop after any leg times out. The three timeout variables allow you to customize the timeout settings for the various call legs. Use call_timeout when using the bridge application, and use originate_timeout when using the originate API. Use leg_timeout if you wish to have a different timeout value for each dial string. In that case, use the [leg_timeout=###] square bracket notation for each dial string:

```xml
<action application="bridge" data="[leg_timeout=10]sofia/internal/userA@local.pbx.com|[leg_timeout=15]sofia/internal/userB@local.pbx.com"/>
```

Use originate_continue_on_timeout to force bridge or originate to continue dialing even if one of the endpoints fails with a timeout:

```xml
<action application="bridge" data="{originate_continue_on_timeout=true}[leg_timeout=10]sofia/internal/userA@host|[leg_timeout=15]sofia/internal/userB@host"/>
```

Keep in mind that by default, a **timeout** (that is, a no answer) will end the entire bridge or originate if you do not set originate_continue_on_timeout to true.

Another thing to keep in mind is handling cases where you are calling a phone number that has voicemail. For example, if you are trying to implement a type of "find me, follow me" and one of the numbers being called is a mobile phone with voicemail, you need to decide whether you want that phone's voicemail to answer your call. If it does answer, then the bridge will be completed. If you do not want the voicemail to answer and end the bridge (so that your bridge will keep dialing the remaining endpoints), then be sure to set **leg_timeout** to a relatively low value. If the voicemail picks up after 15 seconds, then you may wish to set **leg_timeout=12**. In most cases, you will need to make several test calls to find the best timeout values for your various endpoints.
Using individual bridge calls

In some cases, you may find that it is helpful to make a dial attempt to a single endpoint and then do some processing prior to dialing the next endpoint. In these cases, the pipe-separated list of endpoints will not suffice. However, the FreeSWITCH XML dialplan allows you to do this in another way. Consider this excerpt:

```xml
<extension name="ring_sequentially">
  <condition field="destination_number" expression="^(2001)$">
    <action application="set" data="continue_on_fail=true"/>
    <action application="set" data="hangup_after_bridge=true"/>
    <action application="bridge" data="{ignore_early_media=true}
                    sofia/internal/userA@local.pbx.com"/>
    <action application="log" data="INFO call to userA failed."/>
    <action application="bridge" data="{ignore_early_media=true}
                    sofia/internal/userB@local.pbx.com"/>
    <action application="log" data="INFO call to userB failed."/>
  </condition>
</extension>
```

Key to this operation are the highlighted lines. In the first of them, we set `continue_on_fail` to `true`. This channel variable tells FreeSWITCH to keep processing the actions in the extension even if a bridge attempt fails. After each bridge attempt, you can do some processing. Note, however, that we set `hangup_after_bridge` to `true`. This is done so that the dialplan does not keep processing after a successful bridge attempt (for example, if the call to userA was successful, we would not want to call userB after userA hung up). You may add as many additional bridge endpoints as you need.

See also

- The Ringing multiple endpoints simultaneously and Advanced multiple endpoint calling with enterprise originate recipe in this chapter
Advanced multiple endpoint calling with enterprise originate

You've seen many ways of ringing multiple destinations with many options, but in some cases even this is not good enough. Say you want to call two destinations at once, but each of those two destinations is a separate set of simultaneous or sequential destinations.

For instance, you want to call Bill and Susan at the same time, but Bill prefers that you try his cell first, and then try all of his landlines at the same time. Susan, however, prefers that you call her desk first, then her cell, and finally her home. This is a complicated problem, and the solution to it is called enterprise originate. The term "enterprise" is used to indicate an increased level of indirection, dimension, or scale. Basically, you are doing everything the originate syntax has to offer, but you are doing entire originates in parallel in a sort of "super originate".

Getting ready

The first thing you need to do to take advantage of enterprise originate is to fully understand regular originate. Originatel is the term used to indicate making an outbound call. Although there is an originate command that can be used at fs_cli, the method by which you mostly use the originate command is with the bridge dialplan application.

The bridge application versus the originate command

Why do we talk about a regular originate when discussing the bridge application? Are the bridge application and the originate command not completely different? No! This is a common misconception. The bridge application is used in the dialplan, but it does exactly the same thing that the originate command does—it creates a new call leg. In fact, bridge and originate use exactly the same code in the FreeSWITCH core. The only difference between the two is where they are used. The originate command is used in fs_cli to create a new call leg. The bridge application is used in the dialplan to create a new call to which an existing call leg can be connected or bridged.

You will need to open conf/dialplan/default.xml in a text editor or edit a new XML file in the conf/dialplan/default/subdirectory.
How to do it...

The usage of enterprise originate is similar to the \textit{ring simultaneously} example, but an alternate delimiter (\texttt{:_:}) is used:

```
<extension name="enterprise_originate">
  <condition field="destination_number" expression="^(2000)$">
    <action application="bridge"
      data="{ignore_early_media=true}sofia/internal/
        userA@local.pbx.com:_:{myoption=true}sofia/internal/
        userB@local.pbx.com"/>
  </condition>
</extension>

<extension name="enterprise_originate2">
  <condition field="destination_number" expression="^(2001)$">
    <action application="bridge"
      data="{ignore_early_media=true}sofia/internal/
        userA@local.pbx.com,sofia/internal/
        userB@local.pbx.com:_:sofia/internal/
        userC@local.pbx.com,sofia/internal/userD@local.pbx.com"/>
  </condition>
</extension>
```

How it works...

The entire input string is broken down into smaller strings based on the \texttt{:_:} symbol.

Each of those smaller strings is fed to the regular originate engine in parallel, and the first channel to answer will be bridged to the caller. Once one endpoint answers, the rest of the calls in the enterprise will be \textit{canceled}.

There's more...

Enterprise originate has a few special aspects to consider when using it to place calls.
Routing Calls

Setting variables in enterprise originate

As you know, you can use the \{var=val\} syntax to define special variables to be set on all the channels produced by originate, and \[var=val\] to define variables per leg in a call with many simultaneous targets. Enterprise originate uses these as well, but remember that each string separated by the :_:_ delimiter is its own self-contained instance of originate, so \{var=val\} becomes local only to that single originate string. If you want to define variables to be set on every channel of every originate, you must define them at the very beginning of the string, using the \<var=val\> notation. This indicates that you should pass these variables to every leg inside every originate. Consider the following enterprise originate:

```
<action application="bridge" data="<ignore_early_media=true>
  \{myvar=inner1\}[who=userA]sofia/internal/userA@local.pbx.com,
  [who=userB]sofia/internal/userB@local.pbx.com:_:{myvar=inner2} 
  [who=userC]sofia/internal/userC@local.pbx.com,[who=userD]sofia/ 
  internal/userD@local.pbx.com"/>
```

At first glance, this may seem confusing, but when you break it down, you can see what the values of the variables are for each channel. This table shows the values:

<table>
<thead>
<tr>
<th>Channel</th>
<th>$[ignore_early_media]</th>
<th>$[myvar]</th>
<th>$[who]</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:userA@local.pbx.com">userA@local.pbx.com</a></td>
<td>true</td>
<td>inner1</td>
<td>userA</td>
</tr>
<tr>
<td><a href="mailto:userB@local.pbx.com">userB@local.pbx.com</a></td>
<td>true</td>
<td>inner1</td>
<td>userB</td>
</tr>
<tr>
<td><a href="mailto:userC@local.pbx.com">userC@local.pbx.com</a></td>
<td>true</td>
<td>inner2</td>
<td>userC</td>
</tr>
<tr>
<td><a href="mailto:userD@local.pbx.com">userD@local.pbx.com</a></td>
<td>true</td>
<td>inner2</td>
<td>userD</td>
</tr>
</tbody>
</table>

Once you know which syntax to use, it becomes a simple thing to set the channel variables for individual legs inside originates, or the entire enterprise originate.

Ringback

Unlike the regular originate, signaling cannot be passed back from one of the inner originates, because there are too many call paths open to properly handle it. Therefore, when using bridge with enterprise originate, you must define the ringback variable if you want to send a ringtone back to the caller.

See also

To learn more about originate and enterprise originate, look at some other examples in this chapter and study the default dialplan distributed with FreeSWITCH. There are several examples of the many things you can do when placing outbound calls found in conf/dialplan/default.xml.
**Time-of-day routing**

It is common for routing of calls to be different, depending on the time of day or day of the week. The FreeSWITCH XML dialplan has a number of parameters to allow this functionality.

### Getting ready

Determine the parameters for your routing. In this example, we will define business hours as Monday through Friday from 8:00 a.m. to 5:00 p.m. Additionally, we will include a `day_part` variable to reflect morning (midnight to noon), afternoon (noon to 6:00 p.m.), and evening (6:00 p.m. to midnight).

### How to do it...

Start at the beginning of your dialplan by following these steps:

1. Add this extension to the beginning of your context:

   ```xml
   <extension name="Time of day, day of week setup" continue="true">
   <condition wday="2-6" hour="8-16" break="never">
     <action application="set" data="office_status=open" inline="true"/>
     <anti-action application="set" data="office_status=closed" inline="true"/>
   </condition>
   <condition hour="0-11" break="never">
     <action application="set" data="day_part=morning" inline="true"/>
   </condition>
   <condition hour="12-17" break="never">
     <action application="set" data="day_part=afternoon" inline="true"/>
   </condition>
   <condition hour="18-23" break="never">
     <action application="set" data="day_part=evening" inline="true"/>
   </condition>
   </extension>
   ```
2. Later in your dialplan, you can use the **office_status** and **day_part** variables. The **office_status** variable will contain either "open" or "closed", and **day_part** will contain "morning", "afternoon", or "evening". A typical usage would be to play different greetings to the caller, depending on whether or not the office is open. Add these dialplan extensions, which will accomplish the task:

```xml
<extension name="tod route, 5001_X">
    <condition field="destination_number" expression="^(5001)$">
        <action application="execute_extension">
            data="5001_${office_status}"/
        </action>
    </condition>
</extension>

<extension name="office is open">
    <condition field="destination_number" expression="^(5001_open)$">
        <action application="answer"/>
        <action application="sleep" data="1000"/>
        <action application="playback" data="ivr/ivr-good_${day_part}.wav"/>
        <action application="sleep" data="500"/>
    </condition>
</extension>

<extension name="office is closed">
    <condition field="destination_number" expression="^(5001_closed)$">
        <action application="answer"/>
        <action application="sleep" data="1000"/>
        <action application="playback" data="ivr/ivr-good_${day_part}.wav"/>
        <action application="sleep" data="500"/>
    </condition>
</extension>
```

3. Save your XML file and issue the **reloadxml** command at **fs_cli**.
How it works...

The Time of day, day of week setup extension defines two channel variables, namely office_status and day_part. Note the use of inline="true" in our set applications. These allow immediate use of the channel variables in later dialplan condition statements. Every call that hits this dialplan context will now have these two channel variables set (they will also show up in CDR records if you need them). You may have also noticed continue="true" in the extension tag and break="never" in the condition tags. These tell the dialplan parser to keep looking for more matches when it would otherwise stop doing so. For example, without continue="true", when the dialplan matches one of the conditions in the Time of day, day of week setup extension, it stops looking at any more extensions in the dialplan. In a similar way, the break="never" attribute tells the parser to keep looking for more conditions to match within the current extension (by default, when the parser hits a failed condition, it stops processing any more conditions within the current extension).

A detailed discussion of dialplan processing can be found in Packt Publishing's FreeSWITCH 1.2 book.

Our sample extension number is 5001. Note the action it takes:

```xml
<action application="execute_extension"
data="5001_${office_status}"/>
```

This sends the call back through the dialplan looking for a destination_number of 5001_open or 5001_closed. We have defined these destinations with the "office is open" and "office is closed" extensions respectively. Now we can play different greetings to the caller: one when the office is open and a different one when the office is closed. As a nice touch, for all calls, we play a sound file that says "Good morning", "Good afternoon", or "Good evening", depending on what the value in the day_part channel variable is.

The execute_extension and transfer dialplan applications

These two applications tell FreeSWITCH to execute another part of the dialplan. The primary difference is that execute_extension will return after executing another portion of the dialplan, whereas transfer will send control to the target extension. In programming parlance, execute_extension is like a gosub command and transfer is like a goto command. The former comes back, but the latter does not.
There's more...

You may be wondering why we did not simply use a condition to test office_status for the open value, and then use action tags for "office open" and anti-action tags for "office closed". There is nothing preventing us from doing this. However, what if you need to have an office status other than "open" or "closed"? For example, what if you have an office that needs to play a completely different greeting during lunch time? This is difficult to accomplish with only anti-action tags, but with our example, it is very simple. Let's make it a bit more challenging by adding a lunch period that runs from 11:30 a.m. to 12:30 p.m.

We cannot use hour="11.5-12.5", but we do have another value we can test — time-of-day. This parameter lets us define periods in the day at a granularity of minutes, or even seconds. The value range can be 00:00 through 23:59 or 00:00:00 through 23:59:59.

Consider this new Time of day, day of week setup snippet:

```
<extension name="Time of day, day of week setup" continue="true">
  <condition wday="2-6" hour="8-16 break="never">
    <action application="set" data="office_status=open" inline="true"/>
    <anti-action application="set" data="office_status=closed" inline="true"/>
  </condition>
  <condition wday="2-6" time-of-day="11:30-12:30" break="never">
    <action application="set" data="office_status=lunch" inline="true"/>
  </condition>
</extension>
```

Notice that we need to explicitly define the weekend, since we cannot rely on a simple Boolean "open" or "closed" condition. However, we now have a new office_status of "lunch" available to us. We define an additional extension to handle this case:

```
<extension name="office is at lunch">
  <condition field="destination_number" expression="^\d{5}_lunch$">
    Add the specific dialplan actions for handling calls during the office's lunch hour, and you are done. You can add as many new office statuses as you need.

See also

- Refer to the XML dialplan page at http://freeswitch.org/confluence/display/FREESWITCH/XML-Dialplan for more information on the usage of the break, continue, and inline attributes.
Manipulating SIP To: headers on registered endpoints to reflect DID numbers

Sometimes, when routing calls to endpoints that are registered with your system, you would want to utilize custom SIP To: headers. For example, if you are routing DIDs to a PBX or switch, the device you are sending the call to might expect the phone number you wish to reach in the To: SIP header. However, the customer or PBX may have only a single registration to your service that represents multiple DIDs that need to be sent to them.

By default, no flags exist for changing the To: header to match the DID when calling a registered endpoint. Since the registration to your server is typically done via a generic username that is not related to the DID, you must program your dialplan to retrieve a user's registration information and parse out the username portion of the To: header, replacing it with your own. Care must be taken to replace only the username portion and keep the remaining parameters (after @) intact, especially if the NAT traversal is expected to continue operating.

Getting ready

Be sure that you have your DIDs and users configured. In this example, we will use testuser as the username, with a phone number of 4158867999, and our domain will be my.phoneco.test.

How to do it...

Create a dialplan extension specifically for handling calls to the DID number, and use some regular expression syntax to parse out the information. Here is an example:

```xml
<extension name="call_4158867999">
  <condition field="destination_number" expression="^\+?1?4158867999$"/>
  <condition field="${sofia_contact(testuser@local.pbx.com)}" expression="^[^@]+(.*)">%
    <action application="bridge"
      data="sofia/external/4158867999$1"/>
  </condition>
</extension>
```
Routing Calls

How it works...

You would typically send calls to testuser using the bridge command with an argument of user/testuser. In this scenario, however, you would wish to call the testuser's registered endpoint and replace testuser with a phone number, which is 4158867999 in our example. To do this, you must retrieve the testuser's current dial string and remove the username, replacing it with the DID number.

In this example, we leverage the sofia_contact API and some regular expression magic. The first condition simply matches the user's DID phone number. We only want to act if the destination number is 4158867999. The interesting stuff happens in the second condition. The field is `${sofia_contact(testuser@local.pbx.com)}`. By wrapping an API call in ${}, the dialplan literally executes the API and uses the result as the field value. If we go to fs_cli and type sofia_contact testuser@local.pbx.com, we get the result, which is something like this:

```
sofia/external/johndoe@12.34.56.7;fs_nat=yes
```

The ^[^@]+(.*) regular expression pattern is applied against this value. The result is that everything after, and including, the @ sign is placed in the $1 variable. In our example, $1 contains @12.13.56.7;fs_nat=yes. Finally, we execute bridge with the sofia/external/4158867999$1 dial string. With $1 expanded, our destination is as follows:

```
sofia/external/4158867999@12.34.56.7;fs_nat=yes
```

We have successfully replaced testuser with 4158867999, while preserving the necessary IP address and parameters for contacting the server, and sent the call to the proper destination.
Where to buy this book
You can buy FreeSWITCH 1.6 Cookbook from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.