1

Architecture of FreeSWITCH

Welcome to FreeSWITCH! If you are reading this, then, undoubtedly, you are interested in things like telecommunications and Voice over Internet Protocol (VoIP). FreeSWITCH is revolutionary software created during a telephony revolution. Before looking at the architecture of this powerful software, let's take a look at the colorful world of telecommunications. This will help to put FreeSWITCH into perspective.

In this chapter we cover:

* A telephony revolution
* Advantages of FreeSWITCH
* Endpoint and Dialplan modules
* How FreeSWITCH simplifies complex applications like voicemail

# A revolution has begun and secrets have been revealed

How and why the telephone works is a mystery to most people. It has been kept secret for years. We just plugged our phones into the wall and they worked, and most people do just that and expect it to work. The telephony revolution has begun, and we have begun to pry the secrets from the clutches of the legacy of the telephony industry. Now, everyday individuals like you and me are able to build phone systems that outperform traditional phone services and offer advanced features for relatively low cost. Some people even use FreeSWITCH to provide telephone services for making a profit. FreeSWITCH has been designed to make all of this easier, so we will go over the architecture to get a better understanding of how it works.

Do not be concerned if some of these concepts seem unnaturally abstract. Learning telephony takes time, especially VoIP. In fact, we recommend that you read this chapter more than once. Absorb as much as you can on the first pass, then come back after you complete Chapter 5, Understanding the XML Dialplan. You will be surprised at how much your understanding of VoIP and FreeSWITCH has improved. Then come back and skim it a third time after you have completed Chapter 10, Controlling FreeSWITCH Externally, at this point, you will have a firm grasp of VoIP and FreeSWITCH concepts. Give yourself plenty of time to digest all of these strange new concepts, and soon you will find that you are a skilled FreeSWITCH administrator. If you keep at it, you will be rewarded with a meaningful understanding of this strange and wonderful world we call telephony.

Telephones and the systems are very complicated and have evolved over the years into several varieties. The most popular type of phone in the U.K. and the U.S. is the traditional analog phone, which we affectionately refer to as POTS lines or Plain Old Telephone Service. From the traditional Ma Bell phone up to the long-range cordless phones that most of us have today, one thing has remained the same—the underlying technology. In the last 10-15 years, there has been a convergence of technology between computers and telephones that has produced a pair of affordable alternatives to POTS lines—Mobile phones and VoIP phones (also called Internet Phones).

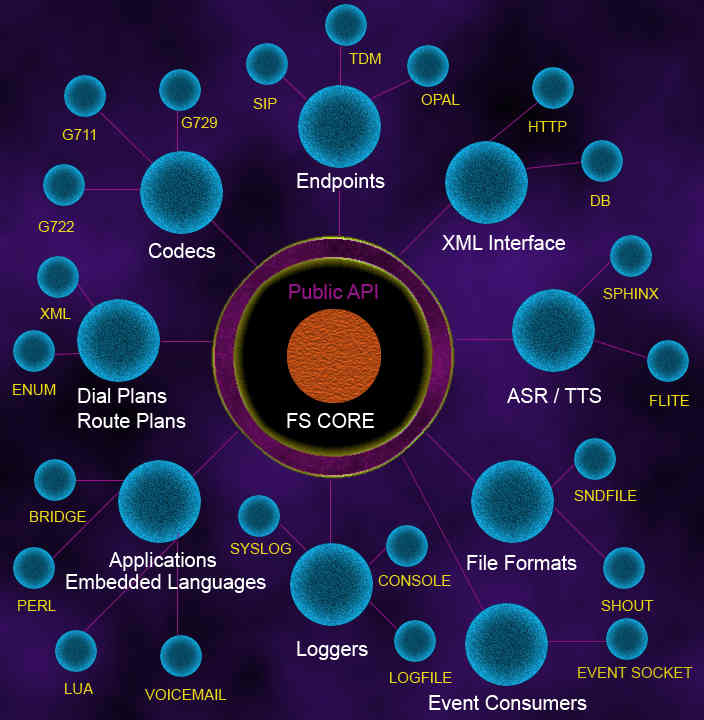
FreeSWITCH fits into this big tangled mess of various telephone technologies by bridging them together, so that they can communicate despite being otherwise completely incompatible. FreeSWITCH also bridges telephone calls with computer programs that you can write yourself, and controls what happens in ways like never before. FreeSWITCH is software that runs on Windows and several UNIX varieties such as Mac OS X, Linux, Solaris, and BSD. This means you can install FreeSWITCH on your home PC or even a high-end server and use it to process phone calls. Installing FreeSWITCH is discussed in detail in Chapter 2, Building and Installation. We will be doing this as soon as we review the basic architecture

# The FreeSWITCH design: modular, scalable, and stable

The design goal of FreeSWITCH is to provide a modular, scalable system around a stable switching core, and provide a robust interface for developers to add to and control the system. Various elements in FreeSWITCH are independent of each other and do not have much knowledge about how the other parts are working, other than what is provided in what are called ''exposed functions''. The functionality of FreeSWITCH can also be extended with loadable modules, which tie a particular external technology into the core.

FreeSWITCH has many different module types that revolve around the central core, much like satellites orbiting a planet. The list includes:

|  |  |
| --- | --- |
| Module Type: | Purpose: |
| Endpoint | Telephone protocols like SIP/H.323 and POTS lines. |
| Dialplan | Parse the call details and decide where to route the call. |
| Codec | Translate between audio formats. |
| Application | Perform a task such as play audio or set data. |
| Application Programming Interface (API) | Export a function that takes text input and returns text output, which could be used across modules or from an external connection. |
| File | Provide an interface to extract and play sound from various audio file formats. |
| Text-To-Speech (TTS) | Interface with text-to-speech engines. |
| Automated Speech Recognition (ASR) | Interface with speech recognition systems. |
| Directory | Connect directory information services, such as LDAP, to a common core lookup API. |
| Chat | Bridge and exchange various chat protocols. |
| Say | String together audio files in various languages to provide feedback to say things like phone numbers, time of day, spell words, and so on. |



Insert Image xxxx\_01\_01.png

By combining the functionality of the various module interfaces, FreeSWITCH can be configured to connect IP phones, POTS lines, and IP-based telephone service. It can also translate audio formats and interfaces with a custom menu system, which you can create by yourself. You can even control a running FreeSWITCH server from another machine. Let's start by taking a closer look at one of the more widely used modules, namely, the Endpoint module.

# Important modules: Endpoint and Dialplan

Endpoint modules are critically important and add some of the key features which make FreeSWITCH the powerful platform it is today. The primary role of these modules is to take certain common communication technologies and normalize them into a common abstract entity which we refer to as a session. A session represents a connection between FreeSWITCH and a particular protocol. There are several Endpoint modules that come with FreeSWITCH, which implement several protocols such as SIP, H.323, Jingle (Google Talk), and some others. We will spend some time examining one of the more popular modules named mod\_sofia.

Sofia-SIP (http://sofia-sip.sourceforge.net) is an open source project sponsored by Nokia, which is determined to make a programming interface to the Session Initiation Protocol or SIP. We use this library in FreeSWITCH in a module we call mod\_sofia. This module registers to all the hooks in FreeSWITCH necessary to make an Endpoint module, and translates the native FreeSWITCH constructs into SIP constructs and back again. Configuration information is taken from the central FreeSWITCH configuration files, which allows mod\_sofia to load user-defined preferences and connection details. This allows FreeSWITCH to accept registration from SIP phones and devices, register to other SIP Endpoints such as service providers, send notifications, and provide services to the phones such as voicemail.

When a SIP call is established between FreeSWITCH and another SIP device, it will show up in FreeSWITCH as an active session. If the call is inbound, it can be transferred or bridged to interactive voice response (IVR) menus, hold music, or one or more extensions, though numerous other options are available. Let's examine a typical scenario where a SIP phone registered as extension 2000 calls extension 2001, with the hope of establishing a call.

First, the SIP phone sends a call setup message to mod\_sofia over the network (mod\_sofia is ''listening'' for such messages). After receiving the message, mod\_sofia in turn parses the relevant details and passes the call into the core state machine in FreeSWITCH. The state machine (in the FreeSWITCH core) then sends the call into the ROUTING state. The next step is to locate the Dialplan module, based on the configuration data for the calling Endpoint. The default and most widely used Dialplan module is the XML Dialplan module. This module is designed to look up a list of instructions from the central XML registry within FreeSWITCH. The XML Dialplan module will parse a series of XML extension objects using regular expression pattern-matching. As we are trying to call 2001, we hope to find an XML extension testing the destination\_number field for something that matches ''2001'' and routes accordingly. The Dialplan is not limited to matching only a single extension. In fact, in Chapter 5, Understanding the XML Dialplan, you will get an expanded definition of the term ''extension''. The XML Dialplan module builds a sort of ''task list'' for the call. Each extension that matches it will have its actions added to the call's task list.

Assuming FreeSWITCH finds at least one extension, the XML Dialplan will insert instructions into the session object with the information it needs to try and connect the call to 2001. Once these instructions are in place, the state of the calling session changes from ROUTING to EXECUTE where the next handler drills down the list and executes the instructions obtained by the ROUTING state. This is where the application interface comes into the picture

Each instruction is added to the session in the form of an application name and a data argument that will be passed to that application. The one we will use in this example is the bridge application. The purpose of this application is to create another session with an outbound connection, then connect the two sessions for direct audio exchange. The argument we will supply to bridge will be user/2001, which is the easiest way to generate a call to extension 2001. A Dialplan entry for ''2001'' might look like this:

<extension name="example">

<condition field="destination\_number " expression="^2001$">

<action application="bridge" data="user/2001"/>

</condition>

</extension>

The extension is named ''example'', and it has a single condition to match. If the condition is matched, it has a single application to execute. In plain language, the mentioned extension could be expressed like this: If the caller dialed ''2001'', then this establishes a connection between the calling party and the endpoint (that is, telephone) at 2001. Consider how this happens.

Once we have inserted the instructions into the session, the session's state will change to EXECUTE, and the FreeSWITCH core will use the data collected to perform the desired action. First, the default execute state handler will parse the command to execute bridge on user/2001, then it will look up the bridge application and pass the user/2001 data in. This will cause the FreeSWITCH core to create a new outbound session of the desired type. User 2001 is also a SIP phone, so the user/2001 will resolve into a SIP dial string, which will be passed to mod\_sofia to ask it to create a new outbound session.

If the setup for that new session is successful, there will now be two sessions in the FreeSWITCH core. The bridge application will take the new session and the original session (the caller's phone) and call the bridge function on it. This allows the audio to flow in both directions, once the person at extension 2001 actually answers the phone. If that user was unable to answer or was busy, a timeout (that is, a failure) would occur and send the corresponding message back to the caller's phone. If a call is unanswered or an extension is busy, many routing options are possible, including call forwarding or voicemail.

All of this happens from the simple action of picking up the phone handset and dialing 2 0 0 1. FreeSWITCH takes all of the complexity of SIP and reduces it to a common denominator. From there, it reduces the complexity further by allowing us to configure a single instruction in the Dialplan to connect the phone at 2000 to the phone at 2001. If we want to allow the phone at 2001 to be able to call the phone at 2000, we can add another entry in the Dialplan going the other way.

<extension name="example 2">

<condition field="destination\_number " expression="^2000$">

<action application="bridge" data="user/2000"/>

</condition>

</extension>

In this scenario, the Endpoint module turned SIP into a FreeSWITCH session and the Dialplan module turned XML into an extension. The bridge application turned the complex code of creating an outbound call, and connecting the audio into a simple application/data pair. Both the Dialplan module and the application module interface are designed around regular FreeSWITCH sessions. Therefore, not only does the abstraction make life easier for us at the user level, it also simplifies the design of the application and the Dialplan because they can be made agnostic of the actual endpoint technology involved in the call. It is because of this abstraction, when we make up a new Endpoint module tomorrow for something like Skype (there is actually such a thing present, by the way), that we can reuse all the same application and Dialplan modules. The same principle applies to the Say, ASR (Automatic Speech Recognition), TTS (Text-to-Speech), and other such modules.

It is possible that you may want to work with some specific data provided by the Endpoint's native protocol. In SIP, for instance, there are several arbitrary headers as well as several other bits of interesting data from the SIP packets. We solve this problem by adding variables to the channel. Using channel variables, mod\_sofia can set these arbitrary values as they are encountered in the SIP data where you can retrieve them by name from the channel in your Dialplan or application. This way we share our knowledge of these special variables with the SIP Endpoint. However, the FreeSWITCH core just sees them as arbitrary channel variables, which the core can ignore. There are also several special reserved channel variables that can influence the behavior of FreeSWITCH in many interesting ways. If you have ever used a scripting language or configuration engine that uses variables, then you are at an advantage because channel variables are pretty much the same concept. There is simply a variable name and a value that is passed to the channel and the data is set.

There is even an application interface for this: the set application that lets you set your own variables from the Dialplan.

<extension name="example 3">

<condition field="destination\_number " expression="^2000$">

<action application="set" data="foo=bar"/>

<action application="bridge" data="user/2000"/>

</condition>

</extension>

The recent example is almost identical to the previous example, but instead of just placing the call, we first set the variable ''foo'' equal to the value ''bar''. This variable will remain set throughout the call and can even be referenced at the end of the call in the detail logs.

The more we build things in small pieces, the more the same underlying resources can be reused, making the system simpler to use. For example, the codec interface knows nothing else about the core, other than its own isolated world of encoding and decoding audio packets. Once a proper codec module has been written, it becomes usable by any Endpoint interface capable of carrying that codec in its audio stream. This means that if we get a Text-To-Speech module working, we can generate synthesized speech on any and all Endpoints that FreeSWITCH supports. It does not matter which one comes first as they have nothing to do with each other. However, the addition of either one instantly adds functionality to the other. The TTS module becomes more useful because it can use more codecs; the codecs have become more useful because we added a new function that can take advantage of them. The same idea applies to applications. If we write a new application module, the existing endpoints will immediately be able to run and use that application.

# Complex applications made simple

FreeSWITCH removes much of the complexity of more advanced applications. Let's look at two examples of a more complex application.

## Voicemail

The first application we will discuss is the voicemail application. The general purpose of this application is probably pretty easy to deduce. It provides voicemail service. This application is useful to add right after the bridge application as a second option, in case the call was not completed. We can do this with a careful combination of application choices, and one of those fancy ''special'' variables that we were discussing earlier. Let's look at a new version of our last extension that also allows us to leave a voicemail:

<extension name="example 4">

<condition field="destination\_number " expression="^2000$">

<action application="set"

data="hangup\_after\_bridge=true"/>

<action application="bridge" data="user/2000"/>

<action application="voicemail"

data="default ${domain} 2000"/>

</condition>

</extension>

Here we see two uses of channel variables. First we set hangup\_after\_bridge=true telling the system to just hang up once we have had at least one successfully bridged call to another phone and to disregard the rest of the instructions. We are also using the domain variable as seen in brackets prefixed with a dollar sign, ${domain}. This is a special variable that defaults to the auto-configured domain name, which all the phones are using from the configuration.

In the example, we check if someone is dialing "2000". We then try to bridge the call to the phone registered to extension 2000. If the call fails or there is no answer, we will continue to the next instruction, which is to execute the voicemail application. We provide the information the application needs to know, and which extension the voicemail is for so it knows how to handle the situation. Next, the voicemail application plays the pre-recorded greeting or generates one for you using the Say module's interface we briefly discussed earlier. It strings together sound files to make a voice say ''The person at extension 2 0 0 0 is not available, please leave a message''. Next, mod\_voicemail prompts you to record a message, and now is your chance to leave your mark in that person's inbox by leaving a voice message. As an additional feature, if you are not satisfied with your recording, you can repeat it as many times as you wish. Once you finally commit, a FreeSWITCH MESSAGE\_WAITING event is fired into the core event system, which is picked up by mod\_sofia by way of an event consumer, where the event information is translated into SIP. If everything goes as planned, the phone registered on extension 2000 will illuminate its message-waiting indicator light!

Again in this example, not only have we seen how to play a greeting, record a message, and send it to a user, we have also uncovered another unsung hero of the FreeSWITCH core—the event system. The FreeSWITCH event system is not a module interface like the other examples, it is a core engine that you can use to bind to named events and react accordingly when an event is received. In other words, throughout the FreeSWITCH core, there are events that are sent and received. Modules can bind to (that is ''listen'' for) various events. They can also ''fire'' events into the event engine; other modules can listen for those events. As we discussed, the Sofia SIP module binds or subscribes to the event designated for MESSAGE\_WAITING information. This allows our mod\_voicemail module to interact with mod\_sofia, without either system having any knowledge about the other's existence. The event is fired by mod\_voicemail, received by mod\_sofia, and translated into the proper SIP message—all seemingly magical, courtesy of the event system.

There are several challenges with such a complex interactive system when considering all of the possible languages it may need to support as well as what files to play for the automated messages and how they are strung together. The Say module supplies a nice way to string files together, but it is somewhat limited to something specific like spelling a word, counting something, or saying a certain date. The way we overcome this is by defining a more complex layer on top of the Say module called Phrase Macros. Phrase Macros are a collection of XML expressions that pull out a list of arguments by matching a regular expression and executing a string of commands. This is very similar to how the XML Dialplan works, only custom-tailored for interactive voice response scenarios (IVR). For example, when mod\_voicemail asks you to record your message, rather than coding in the string of files to make it say that, the code just calls a Phrase Macro called voicemail\_record\_message. This arbitrary string is shared between mod\_voicemail and the Phrase Macro section in the configuration allowing us, the users, to edit the file without doing any fancy programming.

<macro name="voicemail\_record\_message">

<input pattern="^(.\*)$">

<match>

<action function= "play-file"

data="voicemail/vm\_record\_message.wav"/>

</match>

</input>

</macro>

When mod\_voicemail executes the voicemail\_record\_message macro, it first matches the pattern, which, in this case, is just to match everything, as this particular macro has no input. If the macro did have input, the pattern matching could be used to perform different actions based on different input. Once a match is found, the match tag is parsed in the XML for action tags just like in our Dialplan example. This macro just plays the file vm\_record\_message.wav, but more complicated macros, like the ones for verifying your recording or telling you how many messages you have in your inbox, may use combinations of various say modules and playing of audio files. Phrase Macros are discussed in detail in Chapter 6, Using the Built-in XML IVR Engine and used extensively in Chapter 7, Building IVR Applications With Lua.

Once again, we have cooperation among the phrase system, the audio file, and the Say modules loaded by the core being joined together to enable powerful functionality. The Say modules are written specifically for a particular language or ''voice'' within a language. We can programmatically request to say a particular time and have it translated into the proper Say module based on input variables. The Phrase Macro system is a great way to put bigger variable concepts into your code, which can be easily tweaked later by everyday users. For example, if we wanted to make a small IVR that asks us to dial a four-digit number and then just read it back and hang up, we could make one macro called ''mypp\_ask\_for\_digits'' and the other called ''myapp\_read\_digits''. In our code, we would execute these macros by name—the first one when it is time to ask for the digits and the other one to read back the digits by passing in the value we entered. Once this is in place, a less-experienced individual could implement the XML files to play the proper sounds. He or she can use the Say modules to read back the number, and it could all be working in multiple languages with no further coding necessary. Voicemail is just one example of FreeSWITCH in use as an application server. There are endless possibilities when we use FreeSWITCH to connect phone calls with computers.

## Multi-party conferencing

Another popular feature of FreeSWITCH is delivered by the mod\_conference conferencing module. The mod\_conference module provides dynamic conference rooms that can bridge together the audio from several audio channels. This can be used to hold meetings where there are several callers who want to interact on the same call. Each new session that connects to the same conference room will join the others, and instantly be able to talk to all of the other participants at the same time. By using a Dialplan example, similar to the one we used for bridging to another phone, we can make an extension to join a conference room:

<extension name="example 4">

<condition field="destination\_number " expression="^3000$">

<action application="conference" data="3000@default"/>

</condition>

</extension>

This is just as simple as bridging a call, but what is special about this extension is that many callers can call extension 3000 and join the same conference. If three people joined this conference and one of them decides to leave, the other two would still be able to continue their conversation.

The conference module also has other special features, such as the ability to play sound files or text to speech to the whole conference or even just to a single member of the conference. As you may have guessed, we are able to do this by using the TTS and sound file interfaces provided by their respective modules. Once again, the smaller pieces come together to extend the functionality without needing explicit knowledge of the other components in the system.

The conference module also uses the event system in a special way, employing what are called custom events. When it first loads, a module such as mod\_conference can reserve a special event namespace called a subclass. When something interesting happens, such as when a caller joins or leaves a conference, it fires those events on the CUSTOM event channel in the core. When we are interested in receiving such events, all we have to do is subscribe to the CUSTOM event supplying an extra subclass string, which specifies the specific CUSTOM events we are interested in. In this case, it is conference::maintenance. This makes it possible to look out for important things such as when someone joins or leaves the conference or even when they start and stop talking. Conferencing is discussed in detail in Chapter 14, Advanced Features and Further Reading.

## The FreeSWITCH API (FSAPI)

Another very powerful module interface in FreeSWITCH is the FSAPI module. The principle of this type of interface is very simple—it takes a single string of text as input, which may or may not be parsed, especially by the following code. The return value is also a string that can be of any size, from a single character up to several pages of text, depending on the function that was called. One major benefit of FSAPI functions is that a module can use them to call routines in another module, without directly linking into the actual code. The command-line interface of FreeSWITCH or CLI uses FSAPI functions to pass commands from your prompt.

Here is a small example of how we can execute the status FSAPI command from the FreeSWITCH CLI:

freeswitch@internal> status

UP 0 years, 5 days, 23 hours, 32 minutes, 8 seconds, 501 milliseconds, 686 microseconds

FreeSWITCH is ready

87 session(s) since startup

0 session(s) 0/30

1000 session(s) max

min idle cpu 0.00/100.00

Current Stack Size/Max 240K/8192K

What's really happening here is that when we type status and press the Enter key, the word ''status'' is used to look up the status FSAPI function from the module in which it is implemented. The underlying function is then called, and the core is queried for its status message. Once the status data is obtained, the output is written to a stream that comes back and prints as the result of the command.

We have already learned that a module can create and export FSAPI functions, which can be executed from anywhere such as the CLI. But wait, there's more! Modules can also be written to push commands into the FSAPI interface, and send the result over a specific protocol. There are two modules included in FreeSWITCH that do just that— mod\_xml\_rpc and mod\_event\_socket (discussed respectively in Chapter 8, Moving Beyond the Static XML Configuration and Chapter 9, Controlling FreeSWITCH Externally). Consider the example of mod\_xml\_rpc. This module implements the standardized XML-RPC protocol as a FreeSWITCH module. Clients using an XML-RPC interface can connect to FreeSWITCH and execute any FSAPI command they choose. So a remote client could execute an RPC call to status, and get a similar status message to the one we saw in the previous example. This same module also provides FreeSWITCH with a general web server, which allows FSAPI commands to be accessed with a direct URL hit. For example, one could point a browser to http://example.freeswitch.box:8080/api/status to access the status command directly over the World Wide Web. By using this technique, it's possible to create FSAPI commands that work similar to a CGI, providing a dynamic web application that has direct access to FreeSWITCH internals.

As we can see, the FSAPI interface is very versatile. Now we know it can be used to provide a CLI interface, a way for modules to call functions from each other and a way to export WWW or XML-RPC functions. There is still one more use for FSAPI functions that we have not covered. We touched briefly on the concept of channel variables earlier, noting that we can use the expression ${myvariable} to get the value of a certain variable. FSAPI functions can also be accessed this way in the format ${myfunction()}. This notation indicates that the FSAPI command ''myfunction'' should be called, and that the expression should be replaced with the output of that function call. Therefore, we can use ${status()} anywhere when variables are expanded to gain access to the status command. For example:

<action application="set" data="my\_status=${status}"/>

The value placed in the my\_status variable will be the output from the status command.

The drawback to all the versatility provided by a single module interface is that in order to achieve all of this, we have to ''loosely type'' the functionality. This means that there are several cases where a single FSAPI command could easily be accessed, using all of the ways we have discussed. In addition, there are also some other specific functions that are specifically designed for a particular access method. For instance, if we made an FSAPI command that produced HTML intended to be accessed with a web browser, we would not want to access it from the CLI or by referencing it as a variable. Similarly, if we made an FSAPI function that computed some kind of value based on call details, which was designed to be used from the Dialplan, it would not be very useful at the CLI or from the Web. So, with great power comes great responsibility, and this is one case where we need to use common sense to decide when and where to use the proper FSAPI functions to get the most out of them.

## The XML Registry

We have now discussed many of the fundamental components of the FreeSWITCH core and how they interact with each other. We have seen how the event system can carry information across the core, and how the XML Dialplan can query the XML registry for data. This would be a good time to explain the XML registry a bit more. The XML registry is a centrally managed XML document that holds all of the critical data, which FreeSWITCH needs to operate properly. The initial document is loaded from your hard drive and passed into a special pre-processor. This pre-processor can include other XML documents and other special operations, such as setting global variables, which can be resolved by the pre-processor further down in the document.

Once the entire document and all of the included files are parsed, replaced, and generated into a static XML document, this document is loaded into memory. The XML registry is divided into several sections— configuration, dialplan, directory, and phrases. The core and the modules draw their configuration from the configuration section. The XML Dialplan module draws its Dialplan data from the dialplan section. The SIP authentication, user lookup, and the voicemail module read their account information from the directory section. The Phrase Macros pull their configuration from the phrases section. If we make a change to any of the documents on the disk, we can reload the changes into memory by issuing the reloadxml command from the CLI. (This is an example of using the FSAPI interface to communicate with the FreeSWITCH core.)

## Language modules

One distinct type of module that does not have a direct interface to FreeSWITCH-like files and Endpoints, but still offers an immensely powerful connection to existing technology, is the Language module. Language modules embed a programming language like Lua, JavaScript, Perl, and even C# (using mod\_managed) into FreeSWITCH, and transfer functionality between the core and the language's runtime. This allows things like IVR applications to be written in the embedded language, with a simple interface back to FreeSWITCH for all the heavy lifting. Language modules usually register into the core with the application interface and the FSAPI interface and are executed from the Dialplan. Language modules offer lots of opportunities and are very powerful. Using language modules, you can build powerful voice applications in a standard programming language. In some respects, you can actually control a telephone with a programming language.

## The demonstration configuration

Understanding all of these concepts right off the bat is far from easy, and as the maintainers of the software, we do not expect most people to have everything ''just click''. This is the main reason that every new layer we put on top of the core, makes things simpler and easier to learn. The demonstration configuration of FreeSWITCH is the last line of defense between new users of the software and all of the crazy, complicated, and sometimes downright evil stuff, better known as telephony. We try very hard to save the users from such things.

The main purpose of the demonstration configuration in FreeSWITCH is to showcase all of the hundreds of parameters there are to work with. We present them to you in a working configuration that you could actually leave untouched and play with a bit before venturing into the unknown and trying your own hand at changing some of the options. Think of FreeSWITCH as a Lego set. FreeSWITCH and all of its little parts are like a brand new bucket Lego bricks, with plenty of parts to build anything we can imagine. The demonstration configuration is like the sample spaceship that you find in the instruction booklet. It contains step-by-step instructions on exactly how to build something you know will work. After you pick up some experience, you might start modifying your Lego ship to have extra features, or maybe even rebuild the parts into a car or some other creation. The good news about FreeSWITCH is that it comes out of the box already assembled. Therefore, unlike the bucket of Lego bricks, if you get frustrated and smash it to bits, you can just re-install the defaults and you won't have to build it again from scratch. The demonstration configuration is discussed in Chapter 3, Test Driving the Demonstration Configuration.

Once FreeSWITCH has been successfully built on your system, you simply have to launch the program without changing one line in the configuration file. You will be able to point a SIP telephone or software-based SIP soft phone to the address of your computer and make a test call. If you are brave and have ambition of connecting a traditional analog phone, you may want to get the SIP thing under your belt first. This is because it involves a little more work (including purchasing a hardware card for your computer or a magic device called an ATA—analog telephone adapter).

If you have more than one phone, you should be able to configure them to each having an individual extension in the range 1000-1019, which is the default extension number range that is pre-defined in the demonstration configuration. Once you get both phones registered, you will be able to make calls across them or have them to meet in a conference room in the 3000-3399 range. If you call an extension that is not registered or let the phone ring at another extension for too long, the voicemail application will use the phrase system to indicate that the party is not available, and ask you to record a message. If you dial 5000, you can see an example of the IVR system at work, presenting several menu choices demonstrating various other neat things FreeSWITCH can do out of the box. There are a lot of small changes and additions that can be made to the demonstration configuration while still leaving it intact.

For example, using the pre-processor directives we went over earlier, the demonstration configuration loads a list of files into the XML registry from certain places, meaning that every file in a particular folder will be combined into the final XML configuration document. The two most important points where this takes place are where the user accounts and the extensions in the Dialplan are kept. Each of the 20 extensions that are preconfigured with the defaults are stored into their own file. We could easily create a new file with a single user definition and drop it into place to add another user, and simply issue the reloadxml at the FreeSWITCH CLI. The same idea applies to the example dialplan. We can put a single extension into its own file, and load it into place whenever we want.

# Summary

FreeSWITCH is a complex system of moving parts that is intertwined to produce a solid, stable core with flexible and easy-to-extend add-ons. The core extends its interfaces to modules. These modules simplify the functionality further and extend it up to the user. The modules also can bring outside functionality into FreeSWITCH by translating various communication protocols into a common, well-known format. We looked at the various module types, and demonstrated how they revolve around the core and interact with each other to turn simple abstract concepts into higher-level functionalities. We touched base on a few of the more popular applications in FreeSWITCH—the conferencing and voicemail modules and how they, in turn, make use of other modules in the system without ever knowing it. This agnosticism is accomplished by means of the event system. We also saw how the demonstration configuration provides several working examples, to help take the edge off of an otherwise frightening feat of staring down the business end of a full-featured soft-switch.

Now that we have a general idea of what makes FreeSWITCH tick, we will take a closer look at some of these concepts with some real-world examples for you to try. First we obtain a copy of the source code from the Internet, so we can build the software package and install it. From there, we will test out the configuration, so be sure to get yourself a SIP phone or at least a soft-phone. Once we try a few things, we will dive a litter deeper into how things work and create a few things of our own like an extension or two and an IVR menu. So take a deep breath and get ready to dive into the world of telephony with FreeSWITCH!