Chapter 5. Data Formats and Data Models

If you've done any amount of exploration into the world of APIs, you've likely heard about terms like JSON, XML, or YAML. Perhaps you've heard the terms XSD or YANG. You may have heard the term *markup language* when discussing one of these. But what are these things, and what do they have to do with networking or network automation?

In the same way that routers and switches require standardized protocols in order to communicate, applications need to be able to agree on some kind of syntax in order to exchange data between them. For this, applications can use standard *data formats* like JSON and XML (among others). Not only do applications need to agree on how the data is formatted, but also on how the data is structured. *Data models* define how the data stored in a particular data format is structured.

In this chapter, we'll discuss some of the formats most commonly used with network APIs and automation tools, and how you as a network developer can leverage these tools to accomplish tasks. We'll also briefly discuss data models and their role in network automation.

Introduction to Data Formats

A computer programmer typically uses a wide variety of tools to store and work with data in the programs they build. They may use simple variables (single value), arrays (multiple values), hashes (key-value pairs), or even custom objects built in the syntax of the language they're using.

This is all perfectly standard within the confines of the software being written. However, sometimes a more abstract, portable format is required. For instance, a non-programmer may need to move data in and out of these programs. Another program may have to communicate with this program in a similar way, and the programs may not even be written in the same language, as is often the case with something like traditional client-server communications. For example, many third-party user interfaces (UIs) are used to interface with public cloud providers. This is made possible (in a simplified fashion) thanks to standard data formats. The moral of the story is that we need a standard format to allow a diverse set of software to communicate with each other, and for humans to interface with it.

It turns out there are a few options. With respect to data formats, what we're talking about is text-based representation of data that would otherwise be represented as internal software constructs in memory. All of the data formats that we'll discuss in this chapter have broad support over a multitude of languages and operating systems. In fact, many languages, including Python, which we covered in Chapter 4, have built-in tools that make

it easy to import and export data to these formats, either on the filesystem or on the network.

So as a network engineer, how does all this talk about software impact you? For one thing, this level of standardization is already in place from a raw network protocol perspective. Protocols like BGP, OSPF, and TCP/IP were conceived out of a necessity for network devices to have a single language to speak across a globally distributed system—the internet! The data formats in this chapter were conceived for a very similar reason—to enable computer systems to openly understand and communicate with each other.

Every device you have installed, configured, or upgraded was given life by a software developer that considered these very topics. Some network vendors saw fit to provide mechanisms that allow operators to interact with a network device using these widely supported data formats; others did not. The goal of this chapter is to help you understand the value of standardized and simplified formats like these, so that you can use them to your advantage on your network automation journey.

For example, some configuration models are friendly to automated methods, by representing the configuration model in these data formats like XML or JSON. It is very easy to see the XML representation of a certain data set in Junos, for example:

```
root@vsrx01> show interfaces | display xml

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <interface-information xmlns="http://xml.juniper.net/junos/12.1X47/
    junos-interface" junos:style="normal">
        <physical-interface>
            <name>ge-0/0/0</name>
            <admin-status junos:format="Enabled">up</admin-status>
            <oper-status>up</oper-status>
            <local-index>134</local-index>
            <snmp-index>507</snmp-index>
            <link-level-type>Ethernet</link-level-type>
            <mtu>1514</mtu>
            <source-filtering>disabled</source-filtering>
            <link-mode>Full-duplex</link-mode>
            <speed>1000mbps</speed>

 ... output truncated ...
```

Now, of course this is not very easy on the eyes, but that's not the point. From a programmatic perspective, this is ideal, since each piece of data is given its own easily parseable field. A piece of software doesn't have to guess where to find the name of the interface; it's located at the well-known and documented tag "name". This is key to understanding the different needs that a software system may have when interacting with infrastructure components, as opposed to a human being on the CLI.

When thinking about data formats at a high level, it's important to first understand exactly what we intend to do with the various data formats at our disposal. Each was created for a different use case, and understanding these use cases will help you decide which is appropriate for you to use.

Types of Data

Now that we've discussed the use case for data formats, it's important to briefly talk about what kind of data might be represented by these formats. After all, the purpose of these formats is to communicate things like words, numbers, and even complex objects between software instances. If you've taken any sort of programming course, you've likely heard of most of these.

NOTE

Note that since this chapter isn't about any specific programming languages, these are just generic examples. These data types may be represented by different names, depending on their implementation.

Chapter 4 goes over Python specifically, so be sure to go back and refer to that chapter for Python-specific definitions and usage.

String
: Arguably, the most fundamental data type is the string. This is a very common way of representing a sequence of numbers, letters, or symbols. If you wanted to represent an English sentence in one of the data formats we'll discuss in this chapter, or in a programming language, you'd probably use a string to do so. In Python, you may see `str` or `unicode` to represent these.

Integer
: There are actually a number (get it?) of data types that have to do with numerical values, but for most people the integer is the first that comes to mind when discussing numerical data types. The integer is exactly what you learned in math class: a whole number, positive or negative. There are other data types like float or decimal that you might use to describe non-whole values. Python represents integers using the `int` type.

Boolean
: One of the simplest data types is boolean, a simple value that is either true or false. This is a very popular type used when a programmer wishes to know the result of an operation, or whether two values are equal to each other, for example. This is known as the `bool` type in Python.

Advanced data structures
: Data types can be organized into complex structures as well. All of the formats we'll discuss in this language support a basic concept known as an *array*, or a *list* in some cases. This is a list of values or objects that can be represented and referenced by some kind of index. There are also key-value pairs, known by many names, such as

dictionaries, hashes, hash maps, hash tables, or maps. This is similar to the array, but the values are organized according to key-value pairs, where both the key and the value can be one of several types of data, like string, integer, and so on. An array can take many forms in Python: sets, tuples, and lists are all used to represent a sequence of items, but are different from each other in what sort of flexibility they offer. Key-value pairs are represented by the `dict` type.

This is not a comprehensive list, but covers the vast majority of use cases in this chapter. Again, the implementation-specific details for these data types really depend on the context in which they appear. The good news is that all of the data formats we'll discuss in this chapter have wide and very flexible support for all of these and more.

Throughout the rest of this chapter, we'll refer to data types in monospaced fonts, like `string` or `boolean`, so that it's clear we're referring to a specific data type.

Now that we've established what data formats are all about, and what types of data may be represented by each of them, let's dive in to some specific examples and see these concepts written out.

YAML

If you're reading this book because you've seen some compelling examples of network automation online or in a presentation and you want to learn more, you may have heard of YAML. This is because YAML is a particularly human-friendly data format, and for this reason, it is being discussed before any other in this chapter.

NOTE

YAML stands for "YAML Ain't Markup Language," which seems to tell us that the creators of YAML didn't want it to become just some new markup standard, but a unique attempt to represent data in a human-readable way. Also, the acronym is recursive!

Reviewing YAML Basics

If you compare YAML to the other data formats that we'll discuss like XML or JSON, it seems to do much the same thing: it represents constructs like lists, key-value pairs, strings, and integers. However, as you'll soon see, YAML does this is an exceptionally human-readable way. YAML is very easy to read and write if you understand the basic data types discussed in the last section.

This is a big reason that an increasing number of tools (see Ansible) are using YAML as a method of defining an automation workflow, or providing a data set to work with (like a list of VLANs). It's very easy to use YAML to get from zero to a functional automation workflow, or to define the data you wish to push to a device.

At the time of this writing, the latest YAML specification is YAML 1.2, published at *http://www.yaml.org/*. Also provided on that site is a list of software projects that implement YAML, typically for the purpose of being read in to language-specific data structures and doing something with them. If you have a favorite language, it might be helpful to follow along with the YAML examples in this chapter, and try to implement them using one of these libraries.

Let's take a look at some examples. Let's say we want to use YAML to represent a list of network vendors. If you paid attention in the last section, you'll probably be thinking that we want to use a `string` to represent each vendor name—and you'd be correct! This example is very simple:

```
---
- Cisco
- Juniper
- Brocade
- VMware
```

NOTE

You'll notice three hyphens (---) at the top of every example in this section; this is a YAML convention that indicates the beginning of our YAML document.

The YAML specification also states that an ellipsis (...) is used to indicate the end of a document, and that you can actually have multiple instances of triple hyphens to indicate multiple documents within one file or data stream. These methods are typically only used in communication channels (e.g., for termination of messages), which is not a very popular use case, so we won't be using either of these approaches in this chapter.

This YAML document contains four items. We know that each item is a `string`. One of the nice features of YAML is that we usually don't need quote or double-quote marks to indicate a string; this is something that is usually automatically discovered by the YAML parser (e.g., PyYAML). Each of these items has a hyphen in front of it. Since all four of these strings are shown at the same level (no indentation), we can say that these strings compose a list with a length of 4.

YAML very closely mimics the flexibility of Python's data structures, so we can take advantage of this flexibility without having to write any Python. A good example of this flexibility is shown when we mix data types in this list (not every language supports this):

```
---
- Core Switch
- 7700
- False
- ['switchport', 'mode', 'access']
```

In this example, we have another list, again with a length of 4. However, each item is a totally unique type. The first item, `Core Switch`, is a `string` type. The second, `7700`, is interpreted as an `integer`. The third is interpreted as a `boolean`. This "interpretation" is performed by a YAML interpreter, such as PyYAML. PyYAML, specifically, does a pretty good job of inferring what kind of data the user is trying to communicate.

NOTE

YAML `boolean` types are actually very flexible, and accept a wide variety of values that really end up meaning the same thing when interpreted by a YAML parser.

For instance, you could write `False`, as in the above example, or you could write `no`, off, or even simply `n`. They all end up meaning the same thing: a false boolean value. This is a big reason that YAML is often used as a human interface for many software projects.

The fourth item in this example is actually itself a list, containing three `string` items. We've seen our first example of nested data structures in YAML! We've also seen an example of the various ways that some data can be represented. Our "outer" list is shown on separate lines, with each item prepended by a hyphen. The inner list is shown on one line, using brackets and commas. These are two ways of writing the same thing: a list.

NOTE

Note that sometimes it's possible to help the parser figure out the type of data we wish to communicate. For instance, if we wanted the second item to be recognized as a `string`instead of an `integer`, we could enclose it in quotes (`"7700"`). Another reason to enclose something in quotes would be if a `string` contained a character that was part of the YAML syntax itself, such as a colon (:).

Refer to the documentation for the specific YAML parser you're using for more information on this.

Early on in this chapter we also briefly talked about key-value pairs (or dictionaries, as they're called in Python). YAML supports this structure quite simply. Let's see how we might represent a dictionary with four key-value pairs:

```
---
Juniper: Also a plant
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

Here, our keys are shown as `strings` to the left of the colon, and the corresponding values for those keys are shown to the right. If we wanted to look up one of these values in a

Python program for instance, we would reference the corresponding key for the value we are looking for.

Similar to lists, dictionaries are very flexible with respect to the data types stored as values. In the above example, we are storing a myriad of data types as the values for each key-value pair.

It's also worth mentioning that YAML dictionaries—like lists—can be written in multiple ways. From a data representation standpoint, the previous example is identical to this:

```
---
{Juniper: Also a plant, Cisco: 6500, Brocade: True,
VMware: ['esxi', 'vcenter', 'nsx']}
```

Most parsers will interpret these two YAML documents precisely the same, but the first is obviously far more readable. That brings us to the crux of this argument: if you are looking for a more human-readable document, use the more verbose options. If not, you probably don't even want to be using YAML in the first place, and you may want something like JSON or XML. For instance, in an API, readability is nearly irrelevant—the emphasis is on speed and wide software support.

Finally, you can use a hash sign (#) to indicate a comment. This can be on its own line, or after existing data.

```
---
- Cisco    # ocsiC
- Juniper  # repinuJ
- Brocade  # edacorB
- VMware   # erawMV
```

Anything after the hash sign is ignored by the YAML parser.

As you can see, YAML can be used to provide a friendly way for human beings to interact with software systems. However, YAML is fairly new as far as data formats go. With respect to communication directly between software elements (i.e., no human interaction), other formats like XML and JSON are much more popular and have much more mature tooling that is conducive to that purpose.

Working with YAML in Python

Let's narrow in on a single example to see how exactly a YAML interpreter will read in the data we've written in a YAML document. Let's reuse some previously seen YAML to illustrate the various ways we can represent certain data types:

```
---
Juniper: Also a plant
Cisco: 6500
Brocade: True
```

```
VMware:
  - esxi
  - vcenter
  - nsx
```

Let's say this YAML document is saved to our local filesystem as *example.yml*. Our objective is to use Python to read this YAML file, parse it, and represent the contained data as some kind of variable.

Fortunately, the combination of native Python syntax and the aforementioned third-party YAML parser, PyYAML, makes this very easy:

```
import yaml
with open("example.yml") as f:
    result = yaml.load(f)
    print(result)
    type(result)

{'Brocade': True, 'Cisco': 6500, 'Juniper': 'Also a plant',
'VMware': ['esxi', 'vcenter', 'nsx']}
<type 'dict'>
```

NOTE

The Python snippet used in the previous example uses the `yaml` module that is installed with the `pyyaml` Python package. This is easily installed using `pip` as discussed in Chapter 4.

This example shows how easy it is to load a YAML file into a Python dictionary. First, a context manager is used to open the file for reading (a very common method for reading any kind of text file in Python), and the `load()` function in the `yaml` module allows us to load this directly into a dictionary called `result`. The following lines show that this has been done successfully.

Data Models in YAML

In the introduction to this chapter, we mentioned that data models define the structure for how data is stored in a data format, such as YAML, XML, or JSON. Let's take a look at one of the YAML examples from the previous section and discuss it in the context of data models for YAML.

Let's say we had this data stored in YAML:

```
---
Juniper: vSRX
Cisco: Nexus
Brocade: VDX
VMware: NSX
```

Intuitively, we—as people—can look at this data in YAML and understand that it is a list of vendors and a network product from that vendor. We've mentally created a data model that says each entry in this YAML document should contain a pair of `string` values; the first `string` (the key) is the vendor name, and the second `string` (the value) is the product name. Together, these `strings` form a dictionary of key-value pairs.

However, what if we were working with this (implied) data model and supplied this data instead?

```
---
Juniper: vSRX
Cisco: 6500
Brocade: True
VMware:
  - esxi
  - vcenter
  - nsx
```

This is valid YAML, but it's invalid data. Even if Brocade had a product named "True," most YAML interpreters would (by default) read this data as a `boolean` value instead of a `string`. When our software went to do something with this data, it would expect a `string` and get a `boolean` instead—and that would very likely cause the software to produce incorrect results or even crash.

Data models are a way to define the structure and content of data stored in a data format such as YAML. Using a data model, we could explicitly state that the data in the YAML document must be a key-value list, and that each value must be a `string`.

Unfortunately, YAML does not provide any built-in mechanism for describing or enforcing data models. There are third-party tools (one such example is Kwalify). This is one reason why YAML is very suitable for human-to-machine interaction, but not necessarily as well suited for machine-to-machine interaction.

NOTE

YAML is considered a superset of JSON, a format we'll discuss later in this chapter. In theory, this means that tools for validating a JSON schema—the data model for a JSON document—could also validate a YAML document.

The next data format, XML, offers some features and functionality that make it more suitable for machine-to-machine interaction. Let's take a closer look.

XML

As mentioned in the previous section, while YAML is a suitable choice for human-to-machine interaction, other formats like XML and JSON tend to be favored as the data

representation choice when software elements need to communicate with each other. In this section, we're going to talk about XML, and why it is suitable for this use case.

NOTE

XML enjoys wide support in a variety of tools and languages, such as the LXML library in Python. In fact, the XML definition itself is accompanied by a variety of related definitions for things like schema enforcement, transformations, and advanced queries. As a result, this section will attempt only to whet your appetite with respect to XML. You are encouraged to try some of the tools and formats listed on your own.

Reviewing XML Basics

XML shares some similarities to what we've seen with YAML. For instance, it is inherently hierarchical. We can very easily embed data within a parent construct:

```
<device>
  <vendor>Cisco</vendor>
  <model>Nexus 7700</model>
  <osver>NXOS 6.1</osver>
</device>
```

In this example, the `<device>` element is said to be the root. While spacing and indentation don't matter for the validity of XML, we can easily see this, as it is the first and outermost XML tag in the document. It is also the parent of the elements nested within it: `<vendor>`, `<model>`, and `<osver>`. These are referred to as the *children* of the `<device>` element, and they are considered siblings of each other. This is very conducive to storing metadata about network devices, as you can see in this particular example. In an XML document, there may be multiple instances of the `<device>` tag (or multiple `<device>` elements), perhaps nested within a broader `<devices>` tag.

You'll also notice that each child element also contains data within. Where the root element contained XML children, these tags contain text data. Thinking back to the section on data types, it is likely these would be represented by `string` values in a Python program, for instance.

XML elements can also have attributes:

```
<device type="datacenter-switch" />
```

When a piece of information may have some associated metadata, it may not be appropriate to use a child element, but rather an attribute.

The XML specification has also implemented a namespace system, which helps to prevent element naming conflicts. Developers can use any name they want when creating XML documents. When a piece of software leverages XML, it's possible that the software would

be given two XML elements with the same name, but those elements would have different content and purpose.

For instance, an XML document could implement the following:

```
<device>Palm Pilot</device>
```

This example uses the `<device>` element name, but clearly is being used for some purpose other than representing a network device, and therefore has a totally different meaning than our switch definition a few examples back.

Namespaces can help with this, by defining and leveraging prefixes in the XML document itself, using the `xmlns` designation:

```
<root>
  <e:device xmlns:c="http://example.org/enduserdevices">Palm
Pilot</e:device>
  <n:device xmlns:m="http://example.org/networkdevices">
    <n:vendor>Cisco</n:vendor>
    <n:model>Nexus 7700</n:model>
    <n:osver>NXOS 6.1</n:osver>
  </n:device>
</root>
```

There is much more involved with writing and reading a valid XML document. We recommend you check out the W3Schools documentation on XML.

Using XML Schema Definition (XSD) for Data Models

While YAML has some built-in constructs to help describe the data type within (the use of hyphens and indentation), XML doesn't have those same mechanisms. Many XML parsers don't make the same assumptions that PyYAML and other YAML parsers do, for example.

Recall from the beginning of this chapter that we described data formats as allowing applications—or devices, like network devices—to exchange information in standardized ways. XML is one of these standardized ways to exchange information. However, data formats like XML don't enforce what *kind*of data is contained in the various fields and values. To help ensure the right kind of data is in the right XML elements, we have *XML Schema Definition (XSD)*.

XML Schema Definition allows us to describe the building blocks of an XML document. Using this language, we're able to place constraints on where data should (or should not) be in our XML document. There were previous attempts to provide this functionality (e.g., DTD), but they were limited in their capabilities. Also, XSD is actually written in XML, which simplifies things greatly.

One very popular use case for XSD—or really any sort of schema or modeling language—is to generate source code data structures that match the schema. We can then use that source code to automatically generate XML that is compliant with that schema, as opposed to writing out the XML by hand.

For a concrete example of how this is done in Python, let's look once more at our XML example.

```
<device>
  <vendor>Cisco</vendor>
  <model>Nexus 7700</model>
  <osver>NXOS 6.1</osver>
</device>
```

Our goal is to print this XML to the console. We can do this by first creating an XSD document, then generating Python code from that document using a third-party tool. Then, that code can be used to print the XML we need.

Let's write an XSD schema file that describes the data we intend to write out:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <xs:element name="device">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vendor" type="xs:string"/>
      <xs:element name="model" type="xs:string"/>
      <xs:element name="osver" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

In this schema document, we can see that we are describing that each `<device>` element can have three children, and that the data in each of these child elements must be a `string`. Not shown here but supported in the XSD specification is the ability to specify that child elements are required; in other words, you could specify that a `<device>` element *must* have a `<vendor>` child element present.

We can use a Python tool called `pyxb` to create a Python file that contains class object representations of this schema:

```
~$ pyxbgen -u schema.xsd -m schema
```

This will create *schema.py* in this directory. So, if we open a Python prompt at this point, we can import this schema file and work with it. In the below example, we're creating an instance of the generated object, setting some properties on it, and then rendering that into XML using the `toxml()` function:

```
import schema
dev = schema.device()
dev.vendor = "Cisco"
dev.model = "Nexus"
dev.osver = "6.1"
dev.toxml("utf-8")
'<?xml version="1.0" encoding="utf-
8"?><device><vendor>Cisco</vendor><model>Nexus
</model><osver>6.1</osver></device>'
```

This is just one way of doing this; there are other third-party libraries that allow for code generation from XSD files. Also take a look at `generateDS`, located here: *https://pypi.python.org/pypi/generateDS/*.

NOTE

Some RESTful APIs (see Chapter 7) use XML to encode data between software endpoints. Using XSD allows the developer to generate compliant XML much more accurately, and with fewer steps. So, if you come across a RESTful API on your network device, ask your vendor to provide schema documentation—it will save you some time.

There is much more information about XSD located on the W3Schools site at *https://www.w3.org/standards/xml/schema*.

Transforming XML with XSLT

Given that the majority of physical network devices still primarily use a text-based, human-oriented mechanism for configuration, you might have to familiarize yourself with some kind of template format. There are a myriad of them out there, and templates in general are very useful for performing safe and effective network automation.

The next chapter in this book, Chapter 6, goes into detail on templating languages, especially Jinja. However, since we're talking about XML, we may as well briefly discuss Extensible Stylesheet Language Transformations (XSLT).

XSLT is a language for applying transformations to XML data, primarily to convert them into XHTML or other XML documents. As with many other languages related to XML, XSLT is defined on the W3Schools site, and more information is located here at *http://www.w3schools.com/xsl/*.

Let's look at a practical example of how to populate an XSLT template with meaningful data so that a resulting document can be achieved. As with our previous examples, we'll leverage some Python to make this happen.

The first thing we need is some raw data to populate our template. This XML document will suffice:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<authors>
    <author>
        <firstName>Jason</firstName>
        <lastName>Edelman</lastName>
    </author>
    <author>
        <firstName>Scott</firstName>
        <lastName>Lowe</lastName>
    </author>
    <author>
        <firstName>Matt</firstName>
        <lastName>Oswalt</lastName>
    </author>
</authors>
```

This amounts to a list of authors, each with `<firstName>` and `<lastName>` elements. The goal is to use this data to generate an HTML table that displays these authors, via an XSLT document.

An XSLT template to perform this task might look like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output indent="yes"/>
<xsl:template match="/">
  <html>
  <body>
  <h2>Authors</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th style="text-align:left">First Name</th>
        <th style="text-align:left">Last Name</th>
      </tr>
      <xsl:for-each select="authors/author">
      <tr>
        <td><xsl:value-of select="firstName"/></td>
        <td><xsl:value-of select="lastName"/></td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

A few notes on the above XSLT document:

- First, you'll notice that there is a basic `for-each` construct embedded in what otherwise looks like valid HTML. This is a very standard practice in template

language—the static text remains static, and little bits of logic are placed where needed. You'll see more of this in Chapter 6.

- Second, it's also worth pointing out that this `for-each` statement uses a "coordinate" argument (listed as `"authors/author"`) to state exactly which part of our XML document contains the data we wish to use. This is called XPath, and it is a syntax used within XML documents and tools to specify a location within an XML tree.
- Finally, we use the `value-of` statement to dynamically insert (like a variable in a Python program) a value as text from our XML data.

Assuming our XSLT template is saved as *template.xsl*, and our data file as *xmldata.xml*, we can return to our trusty Python interpreter to combine these two pieces and come up with the resulting HTML output.

```
from lxml import etree
xslRoot = etree.fromstring(open("template.xsl").read())
transform = etree.XSLT(xslRoot)
xmlRoot = etree.fromstring(open("xmldata.xml").read())
transRoot = transform(xmlRoot)
print(etree.tostring(transRoot))

<html><body><h2>Authors</h2><table border="1"><tr bgcolor="#9acd32">
<th style="text-align:left">First Name</th><th style="text-align:left">Last
Name
</th></tr>
<tr><td>Jason</td><td>Edelman</td></tr>
<tr><td>Scott</td><td>Lowe</td></tr>
<tr><td>Matt</td><td>Oswalt</td></tr></table></body></html>
```

This produces a valid HTML table for us, seen in Figure 5-1.

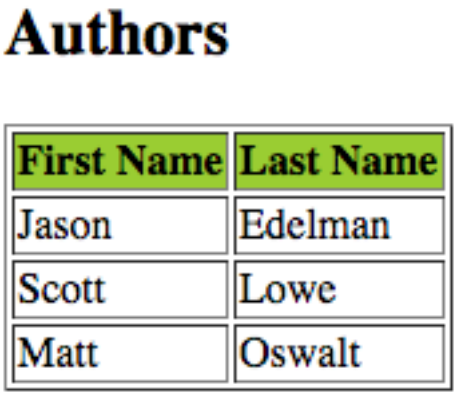

Figure 5-1. HTML table produced by XSLT

XSLT also provides some additional logic statements:

- `<if>`—only output the given element(s) if a certain condition is met
- `<sort>`—sorting elements before writing them as output

- `<choose>`—a more advanced version of the `if` statement (allows "else if" or "else" style of logic)

It's possible for us to take this example even further, and use this concept to create a network configuration template, using configuration data defined in XML, as shown in Examples 5-1 and 5-2:

Example 5-1. XML interface data

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaces>
    <interface>
        <name>GigabitEthernet0/0</name>
        <ipv4addr>192.168.0.1 255.255.255.0</ipv4addr>
    </interface>
    <interface>
        <name>GigabitEthernet0/1</name>
        <ipv4addr>172.16.31.1 255.255.255.0</ipv4addr>
    </interface>
    <interface>
        <name>GigabitEthernet0/2</name>
        <ipv4addr>10.3.2.1 255.255.254.0</ipv4addr>
    </interface>
</interfaces>
```

Example 5-2. XSLT template for router config

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.example.org/routerconfig">

<xsl:template match="/">
    <xsl:for-each select="interfaces/interface">
      interface <xsl:value-of select="name"/><br />
          ip address <xsl:value-of select="ipv4addr"/><br />
    </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

With the XML and XSLT documents shown in Examples 5-1 and 5-2, we can get a rudimentary router configuration in the same way we generated an HTML page:

```
interface GigabitEthernet0/0

ip address 192.168.0.1 255.255.255.0

interface GigabitEthernet0/1

ip address 172.16.31.1 255.255.255.0

interface GigabitEthernet0/2
```

```
ip address 10.3.2.1 255.255.254.0
```

As you can see, it's possible to produce a network configuration by using XSLT. However, it is admittedly a bit cumbersome. It's likely that you will find Jinja a much more useful templating language for creating network configurations, as it has a lot of features that are conducive to network automation. Jinja is covered in Chapter 6.

Searching XML Using XQuery

In the previous section, we alluded to using XPath in our XSLT documents to very particularly locate specific nodes in our XML document. However, if we needed to perform a more advanced lookup, we would need a bit more than a simple coordinate system.

XQuery leverages tools like XPath to find and extract data from an XML document. For instance, if you are accessing the REST API of a router or switch using Python, you may have to write a bit of extra code to get to the exact portion of the XML output that you wish to use. Alternatively, you can use XQuery immediately upon receiving this data to present only the data relevant to your Python program.

XQuery is a powerful tool, almost like a programming language unto itself. For more info on XQuery, check out the W3School specification.

JSON

So far in this chapter, we've discussed YAML, a tool well suited for human interaction and easy import into common programming language data structures. We've also discussed XML, which isn't the most attractive format to look at, but has a rich ecosystem of tools and wide software support. In this section, we'll discuss JSON, which combines a few of these strengths into one data format.

Reviewing JSON Basics

JSON was invented at a time when web developers were in need of a lightweight communication mechanism between web servers and applets embedded within web pages. XML was around at this time, of course, but it proved a bit too bloated to meet the needs of the ever-demanding internet.

You may have also noticed that YAML and XML differ in a big way with respect to how these two data formats map to the data model of most programming languages like Python. With libraries like PyYAML, importing a YAML document into source code is nearly effortless. However, with XML there are usually a few more steps needed, depending on what you want to do.

For these and other reasons, JavaScript Object Notation (JSON) burst onto the scene in the early 2000s. It aimed to be a lightweight version of XML, more suited to the data models found within popular programming languages. It's also considered by many to be more human-readable, although that is a secondary concern for data formats.

NOTE

Note that JSON is widely considered to be a subset of YAML. In fact, many popular YAML parsers can also parse JSON data as if it were YAML. However, some of the details of this relationship are a bit more complicated. See the YAML specification sectionfor more information.

In the previous section, we showed an example of how three authors may be represented in an XML document:

```
<authors>
    <author>
        <firstName>Jason</firstName>
        <lastName>Edelman</lastName>
    </author>
    <author>
        <firstName>Scott</firstName>
        <lastName>Lowe</lastName>
    </author>
    <author>
        <firstName>Matt</firstName>
        <lastName>Oswalt</lastName>
    </author>
</authors>
```

To illustrate the difference between JSON and XML, specifically with respect to JSON's more lightweight nature, here is an equivalent data model provided in JSON:

```
{

    "authors":[
        {
            "firstName": "Jason",
            "lastName": "Edelman"
        },
        {
            "firstName": "Scott",
            "lastName": "Lowe"
        },
        {
            "firstName": "Matt",
            "lastName": "Oswalt"
        }
    ]
}
```

This is significantly simpler than its XML counterpart. No wonder JSON was more attractive than XML in the early 2000s, when "Web 2.0" was just getting started!

Let's look specifically at some of the features. You'll notice that the whole thing is wrapped in curly braces `{}`. This is very common, and it indicates that JSON objects are contained inside. You can think of "objects" as key-value pairs, or dictionaries as we discussed in the section on YAML. JSON objects always use `string` values when describing the keys in these constructs.

In this case, our key is `"authors"`, and the value for that key is a JSON list. This is also equivalent to the list format we discussed in YAML—an ordered list of zero or more values. This is indicated by the square brackets `[]`.

Contained within this list are three objects (separated by commas and a newline), each with two key-value pairs. The first pair describes the author's first name (key of `"firstName"`) and the second, the author's last name (key of `"lastName"`).

We discussed the basics of data types at the beginning of this chapter, but let's take an abbreviated look at the supported data types in JSON. You'll find they match our experience from YAML quite nicely:

Number
: A signed decimal number.

String
: A collection of characters, such as a word or a sentence.

Boolean
: `True` or `False`.

Array
: An ordered list of values; items do not have to be the same type (enclosed in square brackets, `[]`).

Object
: An unordered collection of key-value pairs; keys must be `strings` (enclosed in curly braces, `{}`).

Null
: Empty value. Uses the word `null`.

Let's work with JSON in Python and see what we can do with it. This will be quite similar to what we reviewed when using Python dictionaries in Chapter 4.

Working with JSON in Python

JSON enjoys wide support across a myriad of languages. In fact, you will often be able to simply import a JSON data structure into constructs of a given language, simply with a one-line command. Let's take a look at some examples.

Our JSON data is stored in a simple text file:

```
{
  "hostname": "CORESW01",
  "vendor": "Cisco",
  "isAlive": true,
  "uptime": 123456,
  "users": {
    "admin": 15,
    "storage": 10,
  },
  "vlans": [
    {
      "vlan_name": "VLAN30",
      "vlan_id": 30
    },
    {
      "vlan_name": "VLAN20",
      "vlan_id": 20
    }
  ]
}
```

Our goal is to import the data found within this file into the constructs used by our language of choice.

First, let's use Python. Python has tools for working with JSON built right in to its standard library, aptly called the `json` package. In this example, we define a JSON data structure (borrowed from the Wikipedia entry on JSON) within the Python program itself, but this could easily also be retrieved from a file or a REST API. As you can see, importing this JSON is fairly straightforward (see the inline comments):

```
# Python contains very useful tools for working with JSON, and they're
# part of the standard library, meaning they're built into Python itself.
import json

# We can load our JSON file into a variable called "data"
with open("json-example.json") as f:
    data = f.read()

# json_dict is a dictionary, and json.loads takes care of
# placing our JSON data into it.
json_dict = json.loads(data)

# Printing information about the resulting Python data structure
print("The JSON document is loaded as type {0}\n".format(type(json_dict)))
print("Now printing each item in this document and the type it contains")
for k, v in json_dict.items():
    print(
        "-- The key {0} contains a {1} value.".format(str(k), str(type(v)))
    )
```

Those last few lines are there so we can see exactly how Python views this data once imported. The output that results from running this Python program is as follows:

```
~ $ python json-example.py



The JSON document is loaded as type <type 'dict'>



Now printing each item in this document and the type it contains

-- The key uptime contains a <type 'int'> value.

-- The key isAlive contains a <type 'bool'> value.

-- The key users contains a <type 'dict'> value.
```

```
-- The key hostname contains a <type 'unicode'> value.

-- The key vendor contains a <type 'unicode'> value.

-- The key vlans contains a <type 'list'> value.
```

NOTE

You might be seeing the unicode data type for the first time. It's probably best to just think of this as roughly equivalent to the `str` (string) type, discussed in Chapter 4.

In Python, the `str` type is actually just a sequence of bytes, whereas unicode specifies an actual encoding.

Using JSON Schema for Data Models

In the YAML section, we explained the idea behind data models, and mentioned that YAML doesn't have any built-in mechanisms for data models. In the XML section, we talked about XSD, which allows us to enforce a schema (or data model) within XML—that is, we can be very particular with the type of data contained within an XML document. What about JSON?

JSON also has a mechanism for schema enforcement, aptly named *JSON Schema*. This specification is defined at *http://json-schema.org/documentation.html*, but has also been submitted as an internet draft.

A Python implementation of JSON Schema exists, and implementations in other languages can also be found.

Before we wrap up this chapter, we want to discuss a way of describing data models that is independent of a particular data format. XSD works only for XML, and JSON Schema works only for JSON. Is there a way to describe a data model that can be used with either XML or JSON? There is indeed, and the next section discusses this solution: YANG.

Data Models Using YANG

Throughout this chapter, we've been discussing data models in the context of specific data formats—for example, how to enforce a data model in XML or JSON. In this section, we'd like to take a step back and look at data models more generically, then conclude with a discussion of using YANG to describe networking-specific data models.

To help solidify the concepts of data models, let's quickly review some key facts about data models:

- Data models describe a constrained set of data in the form of a schema language (like XSD, for example).
- Data models use well-defined types and parameters to have a structured and standard representation of data.
- Data models do not transport data, and data models don't care about the underlying transport protocols in use (for example, you could use JSON over HTTP, or you could use XML over HTTP).

Now that you understand what data models are, we're going to shift focus to YANG specifically.

YANG Overview

YANG is a data modeling language defined in RFC 6020. It is analogous to what we mentioned with respect to XSD for generic XML data, but YANG is specifically focused on network constructs. YANG is used to model configuration and operational state data and also used to model general RPC data. General RPC data and tasks allow us to model generic tasks, such as upgrading a device.

YANG provides the ability to define syntax and semantics to more easily define data using built-in and customizable types. You can enforce semantics such that VLAN IDs must be between 1 and 4094. You can enforce the operational state of an interface in that it must be "up" or "down." The model defines these types of constructs and ultimately becomes the source of truth on what's permitted on a network device.

There are various types of YANG models. Some of these YANG models were created by end users; others were created by vendors or open working groups.

- There are industry standard models from groups like the IETF and the OpenConfig Working Group. These models are vendor and platform neutral. Each model produced by an open standards group is meant to provide a base set of options for a given feature.
- Of course, there also are vendor-specific models. As you know, almost every vendor has their own solution for multichassis link aggregation (VSS, VPC, MC-LAG, Virtual Chassis). This means each vendor would need to have their own model if they adopt a model-driven architecture.
- Per vendor, you may even see differences in a given feature. Thus, there are also platform-specific models. Maybe OSPF operates differently on platform X versus platform Y from the same vendor. This would require a different model.

Taking a Deeper Dive into YANG

There is only so much that can be conveyed with words. If seeing a picture is worth a thousand words, then seeing how YANG maps to XML, JSON, and CLI must also be worth a thousand words. We're going to dive deep into YANG statements to make the point on how YANG translates into data that two systems use to communicate.

NOTE

Note that we are not showing how to create a custom YANG model, as that is out of scope for this book. But we are highlighting a few YANG statements to help you better understand YANG.

The YANG language includes the YANG `leaf` statement, which allows you to define an object that is a single instance, has a single value, and has no children.

```
leaf hostname {

    type string;

    mandatory true;

    config true;

    description "Hostname for the network device";

}
```

You can extrapolate from the YANG `leaf` statement what this code is doing. It's defining a construct that will hold the value of the hostname on a network device. It is called `hostname`, it must be a string, it is required, but it is also configurable.

You can also define operational data with YANG `leaf` statements, setting `config` to be **false**.

A YANG `leaf` is represented in XML and JSON as a single element or key-value pair.

```
<hostname>NYC-R1</hostname>
```

```
{

    "hostname": "NYC-R1"
}
```

Another YANG statement is the `leaf-list` statement. This is just like the `leaf` statement, but there can be multiple instances. Since it's a list object, there is a parameter

called `ordered-by` that can be set to `user` or `system` based on whether ordering is important—that is, ACLs versus SNMP community strings versus name servers.

```
leaf-list name-server {

    type string;

    ordered-by user;

    description "List of DNS servers to query";

}
```

A YANG `leaf-list` statement is represented in XML and JSON as a single element, such as the following (first in XML, then in JSON):

```
<name-server>8.8.8.8</name-server>
<name-server>4.4.4.4</name-server>
```

```
{

    "name-server": [
        "8.8.8.8",
        "4.4.4.4"
    ]
}
```

The next YANG statement we'll look at is the YANG `list`. It allows you to create a list of `leaf`s or `leaf-list`s. Here is an example of a YANG `list` definition.

```
list vlan {

   key "id";

   leaf id {

       type int;

       range 1..4094;

   }

   leaf name {

       type string;
```

```
    }

}
```

More importantly for our context is to understand how this is modeled in XML and JSON. Here are examples of how XML and JSON, respectively, would represent this YANG model.

```
<vlan>
    <id>100</id>
    <name>web_vlan></name>
</vlan>
<vlan>
    <id>200</id>
    <name>app_vlan></name>
</vlan>
```

```
{

    "vlan": [
        {
            "id": "100",
            "name": "web_vlan"
        },
        {
            "id": "200",
            "name": "app_vlan"
        }
    ]
}
```

It should be starting to make more sense how data is modeled in YANG and how it's sent over the wire as encoded XML or JSON.

One final YANG statement is the YANG `container`. Containers map directly to hierarchy in XML and JSON. In our previous example, we had a list of VLANs, but no outer construct or tag element in XML that contained all VLANs. We are going to add a container called `vlans` to depict this.

```
container vlans {

    list vlan {

       key "id";

       leaf id {

           type int;
```

```
            range 1..4094;

        }

        leaf name {

            type string;

        }

    }

}
```

The only difference from our last example was adding in the first `container` statement. Here are the final XML and JSON representations of that complete object:

```
<vlans>
  <vlan>
      <id>100</id>
      <name>web_vlan></name>
  </vlan>
  <vlan>
      <id>200</id>
      <name>app_vlan></name>
  </vlan>
</vlans>
```

```
{

    "vlans": {
        "vlan": [
            {
                "id": "100",
                "name": "web_vlan"
            },
            {
                "id": "200",
                "name": "app_vlan"
            }
        ]
    }
}
```

The point of this section was not just to provide a brief introduction to a few different YANG statements, but to really show that YANG is simply a modeling language. It's a way to enforce constraints on data inputs, and these inputs when being used in an API are

represented as XML and JSON. It could be any other data format as well. The device itself then performs checks to see if data adheres to the underlying model being used.

A modeling language like YANG allows you to define semantics and constraints for a given data set. How does a network device ensure VLANs are *between* 1 and 4094? How does a network device ensure the administrative state is either *shutdown* or *no shutdown*? You answer these types of questions by having proper definitions of data. These definitions are defined in a schema document or a specific modeling language. One option for this is to use XML Schema Definitions, which we reviewed earlier in this chapter. However, XSDs are generic. While they are a good way to define a schema for XML documents, XSDs are not *network smart*, and as a result the industry is seeing a shift with how schemas and models are written. YANG is a modeling language specifically built for networking; it understands networking constructs. As an example, it has built-in types to validate whether an input is a valid IPv4 address, BGP AS, or MAC address. YANG is also neutral to the encoding type. A model can be written in YANG and then be *represented* as either JSON or XML. This is what RESTCONF offers. RESTCONF is a REST API that uses XML- or JSON-encoded data that happens to represent data defined by YANG models. We'll discuss RESTCONF in more detail in Chapter 7.

Summary

In this chapter, we've discussed a few key concepts. *Data formats* such as YAML, XML, and JSON are used to format (or encode) data for exchange between applications, systems, or devices. These data formats specify how data is formatted (or encoded), but don't necessarily define the structure of the data. *Data models* define the structure of data formatted (or encoded) in a data format. Sometimes these data models are format-specific; for example, XSD is specific to XML, and JSON Schema is specific to JSON. Finally, YANG provides a format-independent way to describe a data model that can be represented in XML or JSON.

In the next chapter, we're going to talk in more depth about templates, and how they can be used for network automation.