Understanding Data Formats

3.6.1

Data Formats

Rest APIs, which you'll learn about in the next module, let you exchange information with remote services and equipment. So do interfaces built on these APIs, including purpose-dedicated command-line interface tools and integration software development kits (SDKs) for popular programming languages.

When controlling these APIs through software, it is helpful to be able to receive and transmit information in forms that are standards-compliant, and machine- and human-readable. This lets you:

- Easily use off-the-shelf software components and/or built-in language tools to convert messages into forms that are easy for you to manipulate and extract data from, such as data structures native to the programming language(s) you are using. You can also convert them into other standard formats that you may need for various purposes.
- Easily write code to compose messages that remote entities can consume.
- Read and interpret received messages yourself to confirm that your software is handling them correctly, and compose test messages by hand to send to remote entities.
- More easily detect "malformed" messages caused by transmission or other errors interfering with communication.

Today, the three most popular standard formats for exchanging information with remote APIs are XML, JSON, and YAML. The YAML standard was created as a superset of JSON, so any legal JSON document can be parsed and converted to equivalent YAML, and (with some limitations and exceptions) vice-versa. XML, an older standard, is not as simple to parse, and in some cases, it is only partly (or not at all) convertible to the other formats. Because XML is older, the tools for working with it are quite mature.

Parsing XML, JSON, or YAML is a frequent requirement of interacting with application programming interfaces (APIs). Later in this course, you will learn more about **RE**presentational **S**tate **T**ransfer (REST) APIs. For now, it is sufficient for you to understand that an oft-encountered pattern in REST API implementations is as follows:

1. Authenticate, usually by POSTing a user/password combination and retrieving an expiring token for use in authenticating subsequent requests.
2. Execute a GET request to a given endpoint (authenticating as required) to retrieve the state of a resource, requesting XML, JSON, or YAML as the output format.
3. Modify the returned XML, JSON, or YAML.

4. Execute a POST (or PUT) to the same endpoint (again, authenticating as required) to change the state of the resource, again requesting XML, JSON, or YAML as the output format and interpreting it as needed to determine if the operation was successful.

3.6.2

XML

Extensible Markup Language (XML) is a derivative of Structured, Generalized Markup Language (SGML), and also the parent of HyperText Markup Language (HTML). XML is a generic methodology for wrapping textual data in symmetrical tags to indicate semantics. XML filenames typically end in ".xml".

An Example XML Document

For example, a simple XML document might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Instance list -->
<vms>
  <vm>
    <vmid>0101af9811012</vmid>
    <type>t1.nano</type>
  </vm>
  <vm>
    <vmid>0102bg8908023</vmid>
    <type>t1.micro</type>
  </vm>
</vms>
```

This example simulates information you might receive from a cloud computing management API that is listing virtual machine instances.

XML Document Body

For the moment, ignore the first line of the document, which is a special part known as the prologue (more on this below), and the second line, which contains a comment. The remainder of the document is called the body.

Notice how individual data elements within the body (readable character strings) are surrounded by symmetrical pairs of tags, the opening tag surrounded by `<` and `>` symbols, and the closing tag, which is similar, but with a "/" (slash) preceding the closing tag name.

Notice also that some tag pairs surround multiple instances of tagged data (for example, the `<vm>` and corresponding `<vm>` tags). The main body of the document, as a whole, is always surrounded by an outermost tag pair (for example, the `<vms>...</vms>` tag pair) or root tag pair.

The structure of the document body is like a tree, with branches coming off the root, containing possible further branches, and finally leaf nodes, containing actual data. Moving back up the tree, each tag pair in an XML document has a parent tag pair, and so on, until you reach the root tag pair.

User-Defined Tag Names

XML tag names are user-defined. If you are composing XML for your own application, best-practice is to pick tag names that clearly express the meaning of data elements, their relationships, and hierarchy. Tag names can be repeated as required to enclose multiple data elements (or tagged groupings of elements) of the same type. For example, you could create more `<vm>` tag pairs to enclose additional `<vmid>` and `<type>` groupings.

When consuming XML from an API, tag names and their meanings are generally documented by the API provider, and may be representative of usage defined in a public namespace schema.

Special Character Encoding

Data is conveyed in XML as readable text. As in most programming languages, encoding special characters in XML data fields presents certain challenges.

For example, a data field cannot contain text that includes the `<` or `>` symbols, used by XML to demarcate tags. If writing your own XML (without a schema), it is common to use HTML entity encodings to encode such characters. In this case the characters can be replaced with their equivalent `&lt;` and `&gt;` entity encodings. You can use a similar strategy to represent a wide range of special symbols, ligature characters, and other entities.

Note that if you are using XML according to the requirements of a schema, or defined vocabulary and hierarchy of tag names, (this is often the case when interacting with APIs such as NETCONF) you are not permitted to use HTML entities. In the rare case when special characters are required, you can use the characters' numeric representations, which for the less-than and greater-than signs, are `&#60;` and `&#62;` respectively.

To avoid having to individually find and convert special characters, it is possible to incorporate entire raw character strings in XML files by surrounding them with so-called CDATA blocks. Here is an example:

```
<hungarian_sentence><![CDATA[Minden személynek joga van a neveléshez.]]>
</hungarian_sentence>
```

XML Prologue

The XML prologue is the first line in an XML file. It has a special format, bracketed by `<?` and `?>`. It contains the tag name xml and attributes stating the version and a character encoding. Normally the version is "1.0", and the character encoding is "UTF-8" in most cases; otherwise, "UTF-16". Including the prologue and encoding can be important in making your XML documents reliably interpretable by parsers, editors, and other software.

Comments in XML

XML files can include comments, using the same commenting convention used in HTML documents. For example:

```
<!-- This is an XML comment. It can go anywhere -->
```

XML Attributes

XML lets you embed attributes within tags to convey additional information. In the following example, the XML version number and character encoding are both inside the `xml` tag. However, the `vmid` and `type` elements could also be included as attributes in the `xml` tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Instance list -->
<vms>
  <vm vmid="0101af9811012" type="t1.nano" />
  <vm vmid="0102bg8908023" type="t1.micro"/>
</vms>
```

There are a few things to notice here:

- Attribute values must always be included in single or double quotes.
- An element may have multiple attributes, but only one attribute for each specific name.
- If an element has no content, you can use a shorthand notation in which you put the slash inside the open tag, rather than including a closing tag.

XML Namespaces

Some XML messages and documents must incorporate a reference to specific namespaces to specify particular tagnames and how they should be used in various contexts. Namespaces are defined by the IETF and other internet authorities, by organizations, and other entities, and their schemas are typically hosted as public documents on the web. They are identified by Uniform Resource Names (URNs), used to make persistent documents reachable without the seeker needing to be concerned about their location.

The code example below shows use of a namespace, defined as the value of an xmlns attribute, to assert that the content of an XML remote procedure call should be interpreted according to the legacy NETCONF 1.0 standard. This code-sample shows a NETCONF remote procedure call instruction in XML. Attributes in the opening rpc tag denote the message ID and the XML namespace that must be used to interpret the meaning of contained tags. In this case, you are asking that the remote entity kill a particular session. The NETCONF XML schema is documented by IETF.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <kill-session>
    <session-id>4</session-id>
  </kill-session>
</rpc>
```

Interpreting XML

In the above example, what is represented is intended as a list or one-dimensional array (called 'instances') of objects (each identified as an 'instance' by bracketing tags). Each instance object

contains two key-value pairs denoting a unique instance ID and VM server type. A semantically-equivalent Python data structure might be declared like this:

```
vms [
  {
    {"vmid": "0101af9811012"},
    {"type": "t1.nano"}
  },
  {
    {"vmid": "0102bg8908023"},
    {"type": "t1.micro"}
  }
]
```

The problem is that XML has no way of deliberately indicating that a certain arrangement of tags and data should be interpreted as a list. So we need to interpret the XML-writer's intention in making the translation. Mappings between XML tree structures and more-efficient representations possible in various computer languages require understanding data semantics. This is less true of more-modern data formats, like JSON and YAML, which are structured to map well onto common simple and compound data types in popular programming languages.

In this case, the `<vm>` tags bracketing each instance's data (id and type) are collapsed in favor of using plain brackets ({}) for grouping. This leaves you with a Python list of objects (which you can call 'vm objects,' but which are not explicitly named in this declaration), each containing two sub-objects, each containing a key/value pair.

3.6.3

JSON

JSON, or JavaScript Object Notation, is a data format derived from the way complex object literals are written in JavaScript (which is in turn, similar to how object literals are written in Python). JSON filenames typically end in ".json".

Here is a sample JSON file, containing some key/value pairs. Notice that two values are text strings, one is a boolean value, and two are arrays:

```
{
  "edit-config":
  {
    "default-operation": "merge",
    "test-operation": "set",
    "some-integers": [2,3,5,7,9],
    "a-boolean": true,
    "more-numbers": [2.25E+2,-1.0735],
```

```
    }
}
```

JSON Basic Data Types

JSON basic data types include numbers (written as positive and negative integers, as floats with decimal, or in scientific notation), strings, Booleans ('true' and 'false'), or nulls (value left blank).

JSON Objects

As in JavaScript, individual objects in JSON comprise key/value pairs, which may be surrounded by braces, individually:

```
{"keyname": "value"}
```

This example depicts an object with a string value (for example, the word 'value'). A number or Boolean would not be quoted.

JSON Maps and Lists

Objects may also contain multiple key/value pairs, separated by commas, creating structures equivalent to complex JavaScript objects, or Python dictionaries. In this case, each individual key/value pair does not need its own set of brackets, but the entire object does. In the above example, the key "edit-config" identifies an object containing five key/value pairs.

JSON compound objects can be deeply-nested, with complex structure.

JSON can also express JavaScript ordered arrays (or 'lists') of data or objects. In the above example, the keys "some-integers" and "more-numbers" identify such arrays.

No Comments in JSON

Unlike XML and YAML, JSON does not support any kind of standard method for including unparsed comments in code.

Whitespace Insignificant

Whitespace in JSON is not significant, and files can be indented using tabs or spaces as preferred, or not at all (which is a bad idea). This makes the format efficient (whitespace can be removed without harming meaning) and robust for use on command-lines and in other scenarios where whitespace characters can easily be lost in cut-and-paste operations.

3.6.4

YAML

YAML, an acronym for "YAML Ain't Markup Language", is a superset of JSON designed for even easier human readability. It is becoming more common as a format for configuration files, and particularly for writing declarative automation templates for tools like Ansible.

As a superset of JSON, YAML parsers can generally parse JSON documents (but not vice-versa). Because of this, YAML is better than JSON at some tasks, including the ability to embed JSON directly (including quotes) in YAML files. JSON can be embedded in JSON files too, but quotes must be escaped with backslashes `\"` or encoded as HTML character entities `&quote;`

Here is a version of the JSON file from the JSON subsection, expressed in YAML. Use this as an example to understand how YAML works:

```
---
edit-config:
  a-boolean: true
  default-operation: merge
  more-numbers:
  - 225.0
  - -1.0735
  some-integers:
  - 2
  - 3
  - 5
  - 7
  - 9
  test-operation: set
...
```

YAML File Structure

As shown in the example, YAML files conventionally open with three dashes (`---` alone on a line) and end with three dots (... also alone a line). YAML also accommodates the notion of multiple "documents" within a single physical file, in this case, separating each document with three dashes on its own line.

YAML Data Types

YAML basic data types include numbers (written as positive and negative integers, as floats with a decimal, or in scientific notation), strings, Booleans (true and false), or nulls (value left blank).

String values in YAML are often left unquoted. Quotes are only required when strings contain characters that have meaning in YAML. For example, { ,a brace followed by a space, indicates the beginning of a map. Backslashes and other special characters or strings also need to be considered. If you surround your text with double quotes, you can escape special characters in a string using backslash expressions, such as \n for newline.

YAML also offers convenient ways of encoding multi-line string literals (more below).

Basic Objects

In YAML, basic (and complex) data types are equated to keys. Keys are normally unquoted, though they may be quoted if they contain colons (:) or certain other special characters. Keys also do not need to begin with a letter, though both these features conflict with the requirements of most programming languages, so it is best to stay away from them if possible.

A colon (:) separates the key and value:

```
my_integer: 2
my_float: 2.1
my_exponent: 2e+5
'my_complex:key' : "my quoted string value\n"
0.2 : "can you believe that's a key?"
my_boolean: true
my_null: null # might be interpreted as empty string, otherwise
```

YAML Indentation and File Structure

YAML does not use brackets or containing tag pairs, but instead indicates its hierarchy using indentation. Items indented below a label are "members" of that labeled element.

The indentation amount is up to you. As little as a single space can be used where indentation is required, though a best-practice is to use two spaces per indent level. The important thing is to be absolutely consistent, and to use spaces rather than tabs.

Maps and Lists

YAML easily represents more complex data types, such as maps containing multiple key/value pairs (equivalent to dictionaries in Python) and ordered lists.

Maps are generally expressed over multiple lines, beginning with a label key and a colon, followed by members, indented on subsequent lines:

```
mymap:
  myfirstkey: 5
  mysecondkey: The quick brown fox
```

Lists (arrays) are represented in a similar way, but with optionally-indented members preceded by a single dash and space:

```
mylist:
  - 1
  - 2
  - 3
```

Maps and lists can also be represented in a so-called "flow syntax," which looks very much like JavaScript or Python:

```
mymap: { myfirstkey: 5, mysecondkey: The quick brown fox}
mylist: [1, 2, 3]
```

Long Strings

You can represent long strings in YAML using a 'folding' syntax, where linebreaks are presumed to be replaced by spaces when the file is parsed/consumed, or in a non-folding syntax. Long strings cannot contain escaped special characters, but may (in theory) contain colons, though some software does not observe this rule.

```
mylongstring: >
  This is my long string
  which will end up with no linebreaks in it
myotherlongstring: |
  This is my other long string
  which will end up with linebreaks as in the original
```

Note the difference in the two examples above. The greater than (>) indicator gives us the folding syntax, where the pipe (|) does not.

Comments

Comments in YAML can be inserted anywhere except in a long string literal, and are preceded by the hash sign and a space:

```
# this is a comment
```

More YAML Features

YAML has many more features, most often encountered when using it in the context of specific languages, like Python, or when converting to JSON or other formats. For example, YAML 1.2 supports schemas and tags, which can be used to disambiguate interpretation of values. For example, to force a number to be interpreted as a string, you could use the !!str string, which is part of the YAML "Failsafe" schema:

```
mynumericstring: !!str 0.1415
```

3.6.5

Parsing and Serializing

Parsing means analyzing a message, breaking it into its component parts, and understanding their purposes in context. When messages are transmitted between computers, they travel as a stream of characters, which is effectively a string. This needs to be parsed into a semantically-equivalent data-structure containing data of recognized types (such as integers, floats, strings, etc.) before the application can interpret and act upon the data.

Serializing is roughly the opposite of parsing. To communicate information with a REST interface, for example, you may be called upon to take locally-stored data (e.g., data stored in Python dictionaries) and output this as equivalent JSON, YAML, or XML in string form for presentation to the remote resource.

An Example

For example, imagine you wanted to send an initial query to some remote REST endpoint, inquiring about the status of running services. To do this, typically, you would need to authenticate to the REST API, providing your username (email), plus a permission key obtained via an earlier transaction. You might have stored username and key in a Python dictionary, like this:

```
auth = {
  "user": {
    "username": "myemail@mydomain.com",
    "key": "90823ff08409408aebcf4320384"
  }
}
```

But the REST API requires these values to be presented as XML in string form, appended to your query as the value of a key/value pair called "auth":

```
https://myservice.com/status/services?auth=<XML string containing username and key>
```

The XML itself might need to take this format, with Python key values converted to same-name tag pairs, enclosing data values:

```
<user>
  <username>myemail@mydomain.com</username>
  <key>90823ff08409408aebcf4320384</key>
</user>
```

You would typically use a serialization function (from a Python library) to output your auth data structure as a string in XML format, adding it to your query:

```
import dicttoxml    # serialization library
import requests     # http request library
auth = {            # Python dict, containing authentication info
  "user": {
    "username": "myemail@mydomain.com",
    "key": "90823ff08409408aebcf4320384"
  }
}
get_services_query = "https://myservice.com/status/services"
xmlstring = dicttoxml(auth)       # convert dict to XML in string form
myresponse = requests.get(get_services_query,auth=xmlstring)  # query service
```

At this point, the service might reply, setting the variable myresponse to contain a string like the following, containing service names and statuses in XML format:

```
<services>
  <service>
    <name>Service A</name>
    <status>Running</status>
  </service>
  <service>
    <name>Service B</name>
    <status>Idle</status>
  </service>
</services>
```

You would then need to parse the XML to extract information into a form that Python could access conveniently.

```
import untangle     # xml parser library
myreponse_python = untangle.parse(myresponse)
print
myreponse_python.services.service[1].name.cdata,myreponse_python.services.service[1].s
```

In this case, the untangle library would parse the XML into a dictionary whose root element (services) contains a list (service[]) of pairs of key/value object elements denoting the name and status of each service. You could then access the 'cdata' value of elements to obtain the text content of each XML leaf node. The above code would print:

```
Service B  Idle
```

Popular programming languages such as Python generally incorporate easy-to-use parsing functions that can accept data returned by an I/O function and produce a semantically-equivalent internal data structure containing valid typed data. On the outbound side, they contain serializers that do the opposite, turning internal data structures into semantically-equivalent messages formatted as character strings.

3.6.6

Lab - Parse Different Data Types with Python

*In this lab, you will use Python to parse each data format in turn: XML, JSON, and YAML. You will walk through code examples and investigate how each parser works.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Parse XML in Python
- Part 3: Parse JSON in Python
- Part 4: Parse YAML in Python

Parse Different Data Types with Python

3.5
Code Review and Testing

3.7
Software Development and Design Summ...