Using JSON Schema

Learn and Apply JSON Schema by Example,
with Javascript (Node.js) and Python Programs

Featuring JSON Schema draft 4

Joe McIntyre

For Liz

Copyright

First edition: July, 2014

Source Code Licenses

The book contains *Javascript*, and *Python* source code. There are external libraries that are used by some of the programs. The following are the licenses that apply to each of these.

Tiny Validator (for v4 JSON Schema)

Tiny Validator (for v4 JSON Schema), packaged in the file *tv4.js*, is made available under a public domain license. The source is available at,

https://github.com/geraintluff/tv4

The license is available at,

https://github.com/geraintluff/tv4/blob/master/LICENSE.txt

jsonschema 2.3.0

A *JSON Schema* library for Python, made available under a *MIT* license. The source is available at,

https://github.com/Julian/jsonschema

The license is available at,

https://github.com/Julian/jsonschema/blob/master/COPYING

jsonlint

A *JSON* parser made available under a *MIT* license. The source is available at,

https://github.com/zaach/jsonlint

The license is available at,

https://github.com/zaach/jsonlint/blob/master/README.md

Book Content

All *Javascript* and *Python* source code in the book text, and in the set of files accompanying the book, is made available under the following *MIT* license.

> The MIT License (MIT)
>
> Copyright (c) 2014 Joe McIntyre
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction,

Preface

Some software technologies find themselves being used in a variety of contexts, useful without being cumbersome. They get widely used, don't get a reputation as being 'force-fit', and endure. *JSON* is emerging as one of those technologies.

In the message exchange space, *JSON* is commonly used in implementing *RESTful* web services. Its broad use reflects the flexibility of *JSON* formatted content to support iterative definition for message exchanges between systems, coexistence with a range of technologies, and ease of understanding.

A broad cross section of software programs include configuration files. Often these configuration files are supplemented by command line argument support, environment variables, and/or integration with external configuration systems. Over time, the complexity of software configuration has grown with the increasing use of networked resources and greater levels of customization for users. This has led to the evolution of configuration definitions to more flexible technologies such as *JSON* and *XML*.

Often the choice of configuration technology is related to the programming language, runtime, or operating system chosen for the program. However, many programs have cross platform requirements (now including supporting cloud deployments), and may have multiple runtimes operating different portions of the overall system. Thus, it is very desirable for the configuration technology to have support across languages and environments. The inclusion of *Javascript* (*Node.js* and web browser uses) and *Python* examples, for multiple operating systems, shows integration of *JSON* and *JSON Schema* with different runtimes.

In addition to configuration files, many programs also have data management requirements. While some programs have requirements suitable for using databases, others have more modest needs and are better suited to the flexibility of using *JSON* files. However, for *JSON* suited programs, as the content expands (both in the structure of the data model as well as the amount of data stored), there is greater risk of data consistency errors. Also, as is the case with many configuration files, updates to the data may be made with text editing tools and are subject to errors – often minor. Finding these errors is often more work than correcting them – a missing comma, misplaced closing } or], or a typo in a keyword. Fortunately, there are two tools that address this well,

- *JSON* syntax checkers, which find syntax errors.
- *JSON Schema*, and its affiliated validation tools, which find content errors.

As *JSON* has gained support across a broad array of programming languages / runtime platforms, the use of *JSON* formatted files for configuration files and similar uses is now an available option for many projects.

JSON and *JSON Schema* are strong foundational technologies, with many avenues for building interesting and useful capabilities around them. Hopefully the examples spur some new design ideas in your projects.

Web Resources

The *Using JSON Schema* website (*http://usingjsonschema.com*) provides download facilities for the book materials, links to the source code projects, and ongoing information on the topics covered in this book.

The *JSON Validate* website (*http://jsonvalidate.com*) hosts the online version of the validation tool provided.

Source code for all programs and examples is hosted on *GitHub*, under the *usingjsonschema* project (*https://github.com/usingjsonschema*). A navigation page is also available at *https://usingjsonschema.github.io* for the repositories.

Audiences

For those designing and implementing software, this book provides knowledge of the *JSON* and *JSON Schema* technologies, along with examples of how to incorporate their use in a variety of programs.

For those configuring, deploying, and maintaining software that incorporates *JSON* content, the book covers use of *JSON* and *JSON Schema* content, enabling reading and understanding of content, and utilizing *JSON Schema* to improve validation of data completeness and correctness.

What this Book Covers

The book provides an introduction to *JSON* and *JSON Schema* from a usage point of view. Using practical examples, the technologies are presented as they are incorporated in the examples.

JSON has been a published specification since July 2006. *JSON Schema* however is still going through the specification process, and this book uses *draft-04* which was made available in January 2013 and at the time of publication is the current version. While the next revision of *JSON Schema* will be published, the existing schemas and tools will coexist with the next version for some time.

Required Knowledge, Equipment and Software

No prior experience with *JSON* or *JSON Schema* is required.

The program source code provided uses *Javascript* / *Node.js* and *Python*. Basic familiarity with *Javascript* or *Python* is expected, but general computer programming knowledge will suffice for understanding the program logic. A short introduction to *Node.js* is included in *Appendix C* that covers the additional features it brings to the *Javascript* environment.

To use the technologies presented, a computer running *Windows*, *Linux*, *OS X*, or a *Unix* variant can be used. *JSON* and *JSON Schema* content can be created and edited with any text editor.

- When a web browser or web page is referenced, any current web browser can be used.
- Each source code program is shown using *Javascript* on the *Node.js* platform and using *Python*. Both versions are equivalent in function.
  - Information on downloading and installing *Node.js* is covered in *Appendix C*.
  - Information on downloading and installing *Python* is covered in *Appendix D*.

All the software referenced is available without charge. Use of other software that performs equivalent tasks may have licensing requirements, please check with the supplier of the software.

Acknowledgments

The availability of *JSON* and *JSON Schema* as freely available specifications is credited to the authors, contributors, and supporters of the specifications/drafts, and the *Internet Engineering Task Force* (*IETF*).

The *Tiny Validator (for JSON Schema v4)* by Geraint Luff, and *jsonschema* by Julian Berman, are *JSON Schema* validation libraries referenced throughout the book. The ease of learning, and experimenting with, *JSON Schema* are greatly enhanced by these works. The *jsonlint* library by Zachary Carter provides a consistent parsing interface, allowing parser messages and errors to be presented across browsers.

The Stack Overflow community, for the wealth of useful bits of information that fill in obscure gaps across so many topics. No matter how odd a place you have ended up in, Stack Overflow often proves that someone else beat you there!

Conventions Used in the Book

In technical descriptions and examples, references to technologies, programs, screen items, and similar cases, will use italics to distinguish them. For example, *JSON Schema* uses italics as a reference name.

Text intended for use in an operating system command, text editor or similar use will be shown with a distinct font. An example of *JSON* content follows.

```
{
  "name":"Server 14",
  "address":"192.168.1.24",
  "port":80,
}
```

The text shown is the content only. On the screen, other content such as prompts or the user interface for a text editor will be displayed.

If the content is operating system commands, each line can be entered and the *Enter* key pressed at the end of each line to execute each command. When operating system commands differ between operating systems, the commands will be shown for each operating system. Only use the version relevant for your operating system. Note that the *Linux* version of the command will also be applicable for *OS X* and for *Unix* based operating systems.

Using the Programs Provided

There are two styles for invoking the programs provided. The first style are launched using the *Node.js* or *Python* launcher. The second style are packaged for invocation as programs.

Launching Using Script Syntax

Some of the examples are provided for invocation by the language runtime. The format for the command to run the example is *<runtime> <program> <arguments>* such as,

```
node additionService.js -p=8303
```

For *Javascript* programs, the *Node.js* runtime is used with the *node* command. For some operating systems, *nodejs* will be used instead of *node* for the runtime name (the name *node* overlaps with another program that uses the same name). For example,

```
nodejs additionService.js -p=8303
```

For *Python*, some platforms use *python* for the default version, but also support *python3* to allow specifically running the program with the *Python 3.x* runtime. All programs provided should operate correctly for either version.

For any command that starts with *node*, *nodejs* can be substituted when applicable for your platform.

For any command that starts with *python*, *python3* can be substituted when applicable.

Launching Using Program Syntax

The syntax checker and schema validation programs are packaged as programs that can be invoked from a command prompt or script. In each case, the name will execute a shell script (*Linux*, *OS X*, *Unix*) or command file (*Windows*) that will start the corresponding program. The syntax for the command is <*program*> <*arguments*> such as,

```
validate simpleObjectValid.json simpleObject_schema.json
```

If you only use one of either the *Javascript* or *Python* versions of the programs, then the commands can be used as shown. If you install both the *Javascript* and *Python* versions, then the operating system will execute the first program in its program search path. To execute a specific version, add the letter *n* for the *Javascript* / *Node.js* version or *p* for the *Python* version. For example,

```
validaten simpleObjectValid.json simpleObject_schema.json
```

The above command executes the *Javascript* / *Node.js* version specifically, and the command below executes the *Python* version specifically.

```
validatep simpleObjectValid.json simpleObject_schema.json
```

The version of the *Python* runtime, if you have multiple versions installed, will be determined by the configuration present. For instance, on *Windows*, the system *PATH* environment variable.

Acquiring and Installing the Accompanying Materials

The book contains samples and example programs. The source code for these is available to install locally and access online. Online access to all materials is available through the *Using JSON Schema* page at,

http://usingjsonschema.com

On *GitHub*, a project navigation page is available at,

https://usingjsonschema.github.io

The *GitHub* project repository page can also be accessed at,

https://github.com/usingjsonschema

Instructions for downloading the materials and installing the programs are in *Appendix A*.

The example programs use *Node.js* and *Python* runtime platforms. You can choose to use either, or both. For instructions on installing *Node.js*, see *Appendix C*. For instructions on installing *Python*, see *Appendix D*.

Table of Contents

Copyright....iii
Source Code Licenses....iii
Preface....v
Web Resources....vi
Audiences....vi
What this Book Covers....vi
Required Knowledge, Equipment and Software....vi
Acknowledgments....vii
Conventions Used in the Book....vii
Using the Programs Provided....vii
Acquiring and Installing the Accompanying Materials....viii
Table of Contents....ix
1. Introduction....1
Use of Schema Definitions....1
Benefits of Using a Schema....2
What A Schema Does Not Do....2
Validation as a Process....3
2. JSON....4
Structure of JSON Content....4
JSON Single Object....4
JSON Multiple Object Types....6
JSON Array....7
JSON Named Array....8
JSON Multiple Object Types and Arrays....10
Syntax Checking JSON Content....11
3. JSON Schema....16
Constructing a Schema....16
Structure of a JSON Schema Definition....17
Style....17
Values for the “$schema” Element....17
JSON Schema Empty Structure....18
JSON Schema Examples and Validation Tools....18
JSON Schema for a Simple Object (3A)....21
Content Constraints for the Simple Object (3B)....23

JSON Schema for an Array (3C) 24
JSON Schema for a Named Object (3D) 25
JSON Schema for Mixed Objects and Arrays (3E) 26
Patterns for Properties (3F) 27
Dependencies for Properties (3G) 29
Choices: allOf, oneOf, anyOf (3H) 31
AllOf 31
OneOf 32
AnyOf 34
The Negative Constraint: not 35
Object and Array Constraints (3I) 36
Value Constraints (3J) 40
JSON References (Internal) (3K) 48
JSON References (External) and the id Keyword (3L) 50
4. Conditional Content 60
Mutually Exclusive Properties 60
Dependent Properties 61
Selector Driven Schemas 62
Alternative Implementations for Selector Driven Schemas 65
Uses for the Conditional Content Approaches 66
5. Configuration Files 67
Example Configuration File 67
Programs Consuming the Configuration File 68
Summary 73
6. Simple Data Management 74
Usage Examples 74
Capabilities for Simple Data Management 74
Example: Organization and Employee Data 75
Valid Data, Invalid Cross-Reference 78
Additional Custom Validation 79
Custom Validation Processor 79
Validation Data Examples 86
Persistent State Validation Versus In Flight Validation 87
Growing Into a Database 88
Domain Specific Validators 88
7. Designing Software for JSON Message Exchange 90
Implementing Programs that use JSON Message Exchange 90

Javascript / Node.js Implementation 92
Python Implementation 98
Validation Proxy Server 103
8. Command Line Validation Tool 110
Entry Point: main.js / main.py 110
Validation Processing: validate.js / validate.py 113
Using the Tools in Shells and Scripts 123
9. Designing Software to Use JSON Files 126
Validation in Programs 126
In Memory State of JSON Content 126
Persistent State Choices 127
Program Interaction Models for Persistent Storage 128
Persistent Storage of JSON Content 129
Recovery Enabled File Storage 131
Library: safeFile 133
Example: Using Robust Configuration File Capabilities 146
Multiple Data File Programs 149
Appendix A: Installing the Book Materials 150
Installing the Syntax and Validation Tools 150
Using Git to Access the Projects 150
Appendix B: Resources 152
JSON Schema Resources 152
ECMAScript Resources 152
Postal Address Resources 152
Appendix C: Node.js Installation and Introduction 153
Node.js Installation 153
Introduction to Node.js 154
Appendix D: Python Installation 156
Python Package Management 156
About the Author 157
Other Publications by the Author 157

1. Introduction

JSON (*Javascript Object Notation*) and *JSON Schema* provide two essential building blocks for data definition that are easy to use, consume, and evolve. They can be utilized in transient and persistent scenarios, and have a broad range of uses.

Where *JSON* defines the syntax for the data content, *JSON Schema* complements it with content definition. This combination provides an easy to use validation capability that improves ease of use, supports automated testing and content creation, and reduces support complexity.

JSON is usable as a standalone technology, or as a contributing technology in contexts defined by its implementing technology.

For message exchange, *JSON* provides a flexible structure for data packaging. This allows data definitions that can address the a broad range of content. From free-form exchanges, such as web content, to formal data definitions suited for exchange of commercial data.

In the software space, configuration files (and similar storage alternatives) are a staple for many software programs, providing customization capability that is easy to understand, describe, and use. There are many formats to choose from, and various decision criteria that guide the choice of format. *JSON* provides a portable, flexible definition, that is supported by a wide variety of platforms, tools and libraries. It also has an advantage in distributed environments, since configuration objects can be exchanged using the same format between systems, even if they are not using the same operating system, platform, or programming language.

In data management scenarios, *JSON* and *JSON Schema* can be used in prototyping and small scale solutions as a standalone technology. However, for production or use as the solution grows, the data definition capability can be reused as the storage functions transition to database or other technologies.

The book will introduce *JSON* and *JSON Schema* technologies, and use an incremental approach to introducing their capabilities. The use of the technologies in different contexts will be provided, both to illustrate the technologies, as well as to describe the value of the technologies in each usage domain.

Use of Schema Definitions

A schema is a vehicle for describing content to determine compliance. It is often used under the covers as part of vetting content for entry into a program by a validation process.

It can also be effectively used as a tool for creating correct content and accelerating the correction process. This is especially true for content created by people or generated through interactive systems, where dynamic processes are part of the content definition.

Sometimes using a schema or other data definition capability is viewed as “locking down” a system, or making it inflexible. However, with the expressiveness of *JSON Schema*, the purpose of the schema is not to limit flexibility, but rather to correctly express only what is required from the data content, create useful notices of corrections required, and leave the remaining content to the programs to interpret.

Benefits of Using a Schema

Validation can be done with program logic, in fact, a significant portion of many programs is used to validate inputs to be processed by the program. This includes start up command processing, reading configuration files, reading data files, receiving messages, and accepting user input.

In each case, the goal of the validation process is to determine whether the input content is correct and complete before invoking further processing. The use of a schema, and a corresponding validation function to apply the schema to the content, allows the validation function to be processed by a purpose built function rather than bespoke coding. The schema definition, being in the domain of the data definition, is more expressive and easier to understand than program logic.

With simplified definition and maintenance, it is more likely that the validation can be,

- More complete, since the time to produce schema content can be less than the time to write program logic to perform this task.
- Easier to read and understand. Reading program logic for validation often requires reading a mix of stream processing, type checking, and expression checking logic. Reading a schema focuses on the data representation, without the other processing logic mixed in.
- Used correctly by others producing content. Often program logic is a black box, or the documentation limited, making determining all valid inputs for particular elements, or whether elements are required or optional, difficult to ascertain without excellent documentation being provided. Providing a schema to other parties makes the task of creating correct content to provide to the program much easier.

Not all content benefits equally from using a schema. A program that only has one or two configuration options doesn't have a significant amount of validation code, and a program that expects free form content may not have enough definition to be very useful. Selecting the programs, and places in the program, where schema definitions can be of benefit is a design choice.

What A Schema Does Not Do

Using a schema replaces some program logic. Specifically that logic that is required to ensure that the content to be processed is complete and correct. However, there are a number of areas that a schema does not cover,

- Elements that rely on knowledge of other systems or processing that occurs within the program. For example, an enumeration within a schema can validate that the value provided for a customer number has the correct number of digits and number range. However, it cannot validate that a customer has valid credit standing at this time, which requires an additional interaction with another system to receive this dynamic status.
- Complex data format representations, or those that require calculation, may be beyond the scope of the constraints defined in the specifications (such as range checks) or regular expression parser. The schema may represent these as a *string* type and provide validation for what can be expressed (e.g., length, number/letter content), while leaving it to the program to provide additional validation logic.
- Domain specific interpretation may require secondary logic to determine validity. For example, a geographical position may be expressed as latitude and longitude, and the schema can verify that

the coordinates provided are valid both in format and for a location on the planet. However, if the coordinate is part of an address, the schema validation will not be able to determine whether the coordinate is within the bounds of the city or not. A secondary validation step could provide this additional validation step.

These limitations express a boundary of the general purpose validation capability. However, they invite thought on how validation can be implemented in stages, rather than a monolithic function.

Validation as a Process

With some context now on the use of schema definitions, role of validation functions, and limitations, validation can now be considered in the context of a process.

The process can consist of any combination of,

- Schema definitions and general purpose schema validation processors to provide definition and automated processing of many aspects of data content validation.
- Specialized secondary processors that augment the general purpose schema validation with domain specific validation functions.
- Bespoke program logic to handle program specific processing related to the content that has passed through the other validation steps.

For example, building on geographical location example in the limitations, the following logic would fit in each of these stages.

- The first stage determines whether a geographical location using latitude and longitude are valid numbers, and that those numbers fall into the range of values for the planet.
- The second stage determines whether the location is within the city boundaries for the business receiving the message. Since city boundaries are not square, and can change over time, this stage will interact with a system that has domain specific information and processing capability to determine this fact and provide it to this validation processor.
- In the third stage, the program itself can incorporate the geographical location in its user interface to show the location the message pertains to on a map display. Given the preceding validation, it is assured that the location will display in a visible location when just the city map is presented.

This book deals primarily with the first stage, the definition and general purpose processing of schema definitions. The second and third stages are implemented by the programs utilizing the result of the first stage processing.

2. JSON

The format for *JSON* content is published as *RFC 7159* by the *IETF* (*Internet Engineering Task Force*) and is available at

http://tools.ietf.org/html/rfc7159

The specification itself can be read reasonably quickly, it contains about 8 pages of technical content. Often, *JSON* related content will reference the prior version of the specification, *RFC 4627*. The differences between the two are small, so information referencing *RFC 4627* can continue to be considered relevant.

Structure of JSON Content

JSON contains object, array, and name-value pair elements. These can all be contained within one another, with the outermost element always being either an object or an array.

An object element is defined using enclosing braces { }.

An array element is defined using enclosing square brackets [ ].

A name-value pair is defined using the pattern *"name":value* where

- *"name"* is a string and includes the enclosing quotes.
- : (colon) separates the name and value.
- *value* can be a string (enclosing quotes), number, boolean (*true* or *false*), *null*, object (enclosing braces { }), or array (enclosing square brackets).

This structure provides a lot of flexibility, as objects, arrays, and name-value pairs can be defined in many ways. The use of name-value pairs also allows for sparse definitions (no positional constraints for placeholders), and variance in content from object to object.

JSON Single Object

The following is *JSON* that contains only one object, describing the server configuration for an IP server.

Directory:chapter2, file: singleObject.json

```
{
  "name":"Server 14",
  "address":"192.168.1.24",
  "port":80,
  "admin":[1080, 1081]
}
```

A *JSON* object is contained within enclosing braces { }. Within the braces, elements are defined with name-value pairs. Each name-value pair is separated by a comma. A name-value pair is made up of three parts, a name enclosed in quotation marks, a colon, and a value. The value can be,

- A text string. Quotation marks are used to enclose the text string content.
- A number. No quotation marks are required (e.g., the value *80* for *"port"* in the example above).
- A boolean, *true* or *false*, without quotation marks.

- The value *null*, without quotation marks.
- An object. The value starts and ends with enclosing braces { }, and follows this definition recursively.
- An array. The value starts and ends with enclosing square brackets [ ], and contains a list of items (e.g., the value *[1080, 1081]* for *admin*).

A program receiving this content is presented with a single object. The following *Javascript* example shows the content being read from the file and the contents being displayed to the console.

Directory: chapter2, file: singleObject.js.

```
/*
 * Read a JSON file with a single unnamed object,
 * and display the values for each element.
 */
// include the Node.js file system module
var fs = require ("fs");

// read the content of the file synchronously
var data = fs.readFileSync ("singleObject.json");

// convert the text into a JSON object
var server = JSON.parse (data);

// display the server name
console.log ("name: " + server.name);
// display the address
console.log ("address: " + server.address);
// display the port number
console.log ("port: " + server.port);
// display the list of admin ports
for (var ctr = 0; ctr < server.admin.length; ctr ++) {
    console.log ("admin: " + server.admin[ctr]);
}
```

To run this program, use the following command in the directory *chapter2* within the directory the examples have been placed in (e.g., *~/bookujs/chapter2* on *Linux* or *c:\bookujs\chapter2* on *Windows*).

```
node singleObject.js
```

The following is the equivalent program written in *Python*.

Directory: chapter2, file: singleObject.py

```
"""
Read a JSON file with a single unnamed object,
and display the values for each element
"""
# import the loads function from the json module
from json import loads

# read the content of the file synchronously
data = open ("singleObject.json", "rU").read ()

# convert the text into a JSON object
server = loads (data)

# display the server name
print ("name: " + server["name"])
# display the address
print ("address: " + server["address"])
# display the port number
print ("port: " + str (server["port"]))
```

```
# display the list of admin ports
for admin in server["admin"]:
    print ("admin: " + str (admin))
```

To run this program, use the following command in the directory *chapter2* within the directory the examples have been placed in (e.g., *~/bookujs/chapter2* on *Linux* or *c:\bookujs\chapter2* on *Windows*).

```
python singleObject.py
```

Content typically has more than just a single type of object, if multiple types of objects are used, then each object type can be named.

JSON Multiple Object Types

Building on the first example, the following *JSON* contains an *HTTP* server definition (server) and a home page definition (homepage).

Directory chapter2, file: multipleObject.json.

```
{
  "server":
  {
    "name":"Server 14",
    "address":"192.168.1.24",
    "port":80,
    "admin":[1080, 1081]
  },
  "homepage":
  {
    "url":"/",
    "page":"public/home.html"
  }
}
```

The differences from the first example are,

- The enclosing braces { } now include the two objects.
- The first object is now defined with a name-value pair, where the name is *"server"* and the value is the original definition (including the enclosing braces { }).
- A comma is added after the closing brace for the *"server"* object, since an additional object is being added following it.
- The new object, *homepage*, is added using a key-value pair consisting of the *homepage* name, and the *homepage* element definitions within enclosing braces { }.

The software program loading this content will access the two objects using the *server* and *homepage* names to reference each individual object.

For instance, in a simple *Javascript* program reading this content from a file using *Node.js* follows.

Directory chapter2, file: multipleObject.js

```
/*
 * Read a JSON file with multiple object types, and
 * display a value from each object type.
 */
var fs = require ("fs");

// read the file synchronously, convert to a JSON object
var content = fs.readFileSync ("multipleObject.json");
var configuration = JSON.parse (content);

// display the port in the server definition
```

```
console.log ("port: " + configuration.server.port);
// display the url in the homepage definition
console.log ("url: " + configuration.homepage.url);
```

To run this program, use the following command in the *chapter2* directory.

```
node multipleObject.js
```

The following is the equivalent program written in *Python*.

Directory: chapter2, file: multipleObject.py

```
"""
Read a JSON file with multiple object types, and
display a value from each object type.
"""
from json import loads

# read the file and convert to a JSON object
data = open ("multipleObject.json", "rU").read ()
configuration = json.loads (data)

# display the port in the server definition
print ("port: " + str (configuration["server"]["port"]))
# display the url in the homepage definition
print ("url: " + configuration[“homepage"]["url"])
```

To run this program, use the following command in the *chapter2* directory.

```
python multipleObject.py
```

For many schema definitions, the content will be a single instance of a group of objects. However, for those definitions that can have multiple instances of a single type, an array definition can be used.

JSON Array

Many uses of *JSON* will require specifying multiple instances of an element. For example, multiple *HTTP* servers may be defined in a configuration. Building on the first example, an array of *HTTP* server definitions is shown below.

Directory chapter2, file: array.json.

```
[
  {
    "name":"Server 14",
    "address":"192.168.1.24",
    "port":80,
    "admin":[1080, 1081]
  },
  {
    "name":"Server 15",
    "address":"192.168.1.31",
    "port":80,
    "admin":[1080, 1081]
  }
]
```

The differences from the first example are,

- Enclosing square brackets [ ] surround the elements in the array (first and last lines).
- A comma is added after the first element, as a separator before the next element is added.
- A second object has been added to the set, with its own content.

When this content is read by a program, the content will be parsed as an array with two elements. An example displaying of an object from each array element is shown next.

Directory: chapter2, file: array.js

```
/*
 * Read a JSON file with multiple array elements, and display a value
 * from each element.
 */
var fs = require ("fs");

// read the file synchronously and convert to a JSON object
var content = fs.readFileSync ("array.json");
var servers = JSON.parse (content);

// display the first server address
console.log ("server: " + servers[0].name + " " + servers[0].address);
// display the second server address
console.log ("server: " + servers[1].name + " " + servers[1].address);
```

To run this program, use the following command in the *chapter2* directory.

```
node array.js
```

The following is the equivalent program written in *Python*.

Directory: chapter2, file: array.py

```
"""
Read a JSON file with multiple array elements, and display a value
from each element.
"""
from json import loads

# read the file and convert to a JSON object
data = open ("array.json", "rU").read ()
servers = loads (data)

# display the first server address
print ("server: " + servers[0]["name"] + " " + servers[0]["address"])
# display the second server address
print ("server: " + servers[1]["name"] + " " + servers[1]["address"])
```

To run this program, use the following command in the *chapter2* directory.

```
python array.py
```

JSON Named Array

The previous examples used an unnamed array. Like objects, an array can be be identified with a name as well. This can be useful if the structure of the content may be extended in the future to support additional array or object types, allowing the original content to be used consistently before and after the changes.

Directory: chapter2, file: namedArray.json.

```
{
  "servers":
  [
    {
      "name":"Server 14",
      "address":"192.168.1.24",
      "port":80,
      "admin":[1080, 1081]
    },
    {
      "name":"Server 15",
      "address":"192.168.1.31",
      "port":80,
```

```
        "admin":[1080, 1081]
      }
    ]
  }
```

The differences in the *JSON* content are,

- The array is now enclosed in an object, so the root element is enclosed in braces { }.
- The *"servers"* name-value pair name is added before the opening square bracket [ ]

When accessing the named array in a program, the array name will be added to the element. The following shows the updated code from the *array.js* example.

Directory: chapter2, file: namedArray.js

```
/*
 * Read a JSON file with multiple named array elements,
 * and display a value from each element.
 */
var fs = require ("fs");

// read the file synchronously, convert to a JSON object
var content = fs.readFileSync ("namedArray.json");
var data = JSON.parse (content);

// display the first server address
console.log ("server");
console.log ("  name: " + data.servers[0].name);
console.log ("  address: " + data.servers[0].address);
// display the second server address
console.log ("\nserver");
console.log ("  name: " + data.servers[1].name);
console.log ("  address: " + data.servers[1].address);
```

In the *console.log* calls at the end of the code, the element references now include the array name from the *JSON* content. Where the first element in *array.js* was *servers[0].name*, since the array was unnamed, now becomes *servers.servers[0].name*, with the named array.

To run this program, use the following command in the *chapter2* directory.

```
node namedArray.js
```

The following is the equivalent program written in *Python*.

Directory: chapter2, file: namedArray.py

```
"""
Read a JSON file with multiple named array elements,
and display a value from each element.
"""
from json import loads

# read the file and convert to a JSON object
content = open ("namedArray.json", "rU").read ()
data = loads (content)

# display the first server address
print ("server")
print ("  name: " + data["servers"][0]["name"])
print ("  address: " + data["servers"][0]["address"])
# display the second server address
print ("\nserver")
print ("  name: " + data["servers"][0]["name"])
print ("  address: " + data["servers"][0]["address"])
```

To run this program, use the following command in the *chapter2* directory.

```
python namedArray.py
```

The next topic describes this in a broader context, combining multiple object types and arrays.

JSON Multiple Object Types and Arrays

Bringing together the previous two concepts, this file example is typical of *JSON* content. Building on the prior examples, the following schema includes a *HTTP* server definition with a set of web pages it hosts.

Directory: chapter2, file: mixed.json

```
{
  "server":
  {
    "name":"Server 14",
    "address":"192.168.1.24",
    "port":80,
    "admin":[1080, 1081]
  },
  "pages":
  [
    {
      "url":"/",
      "page":"public/home.html"
    },
    {
      "url":"/about",
      "page":"public/company/about.html"
    },
    {
      "url":"/careers",
      "page":"public/company/careers.html"
    }
  ]
}
```

In the *JSON* content the *"server"* element is a single object, while the *"pages"* element is an array. The use of an object definition or an array definition for each data type is an independent choice, there can be any number of either included.

For a program using this content, a simple program accessing this data is shown below.

Directory chapter2, file: mixed.js

```
/*
 * Read a JSON file with multiple object types and an array.
 * Display a value from the object type and each array.
 */
var fs = require ("fs");

// read the file synchronously and convert to a JSON object
var content = fs.readFileSync ("mixed.json");
var configuration = JSON.parse (content);

// display the port in the server definition
console.log ("port: " + configuration.server.port);
// display the url of the each cataloged page
for (var ctr = 0; ctr < configuration.pages.length; ctr ++) {
    console.log ("url: " + configuration.pages[ctr].url);
}
```

To run this program, use the following command in the *chapter2* directory.

```
node mixed.js
```

The following is the equivalent program written in *Python*.

Directory: chapter2, file: mixed.py

```
"""
Read a JSON file with multiple object types and an array.
Display a value from the object type and each array.
"""
from json import loads

# read the file and convert to a JSON object
data = open ("mixed.json", "rU").read ()
configuration = loads (data)

# display the port in the server definition
print ("port: " + str (configuration["server"]["port"]))
# display the url of the each cataloged page
for page in configuration["pages"]:
    print ("url: " + page["url"])
```

To run this program, use the following command in the *chapter2* directory.

```
python mixed.py
```

Syntax Checking JSON Content

JSON content is plain text, and its syntax is well defined. This enables validation of *JSON* content to be straightforward, and for any syntax errors to be readily identified. There are a number of syntax checking tools available for *JSON*, including automatic parsing in text editors and integrated development tools, and in browser based tools.

As a *"hello world"* type of program for *JSON*, a syntax checking tool is presented. These introduce the *JSON* parser usage, and the basic structure for the contents of projects used throughout the book.

Syntax Check Tool – Javascript / Node.js Version

A very simple tool follows that displays a success message ("File contains valid JSON content.") if the syntax is valid, and a failure message if it does not (e.g., "Invalid JSON content: Unexpected string").

The *chapter2/nodejs/jsonsyntax* directory contains the main program for a syntax check command line tool written in *Javascript* that runs on the *Node.js* runtime. The full program content, including code, packaging and tests is in the *ujs-jsonsyntax-node* repository. The source code for the syntax checking function itself is shown next.

Directory: chapter2/nodejs/jsonsyntax, file: jsonsyntax.js

The leading section includes the file system module from *Node.js* (*fs*).

```
 1 /**
 2  * Read a file and determine if content is valid JSON syntax.
 3  *
 4  * Usage: jsonsyntax file
 5  *   file   JSON file to check syntax of
 6  *
 7  * Exit code
 8  *   exit code: 0 for success, 1 for failure.
 9  */
10
11 // file system module
12 var fs = require ("fs");
```

The error, *CheckSyntaxError*, is used in exceptions. The passed parameters are stored for use by the exception handling code in the calling function.

```
14 /**
15  * Error definition thrown when an error occurs.
16  * - error code 1: Invalid name
17  * - error code 2: File does not exist
18  * - error code 3: Error reading file
19  * - error code 4: JSON syntax error
20  *
21  * @param {integer} code Error number
22  * @param {string} message Text message, suitable for display
23  */
24 exports.CheckSyntaxError = CheckSyntaxError;
25 function CheckSyntaxError (code, message)
26 {
27     this.name = "CheckSyntaxError";
28     this.code = code;
29     this.message = message;
30 }
```

The *checkSyntax* function validates the input parameter, reads the specified file, and then parses the data from the file using the *JSON* parser. All errors are handled using exceptions, so if the function reaches its end, then the syntax check is successful.

```
32 /**
33  * Check syntax of file passed in command line argument.
34  *
35  * @param {string} file File to check
36  * @throws CheckSyntaxError
37  */
38 exports.checkSyntax = checkSyntax;
39 function checkSyntax (file) {
40     "use strict";
41
42     // verify file provided
43     if ((file === null) || (file === undefined)) {
44         throw new CheckSyntaxError (1, "Invalid name");
45     }
46
47     // verify file exists
48     if (fs.existsSync (file) === false) {
49         throw new CheckSyntaxError (2, "File not found");
50     }
51
52     // read specified file
53     var data = null;
54     try {
55         data = fs.readFileSync (file);
56     } catch (e) {
57         throw new CheckSyntaxError (3, "Error reading file: " + e.message);
58     }
59
60     // parse the data as JSON
61     try {
62         JSON.parse (data);
63     } catch (e) {
64         throw new CheckSyntaxError (4, e.message);
65     }
66 }
```

The last section contains the main function, executed if the program is called from the command line or in a script. It displays a usage message if the command does not include exactly one argument

(*process.argv* includes 2 static arguments, so one program argument brings its length to 3). If the syntax check is successful, it displays a valid message and the program ends with exit code 0 (zero). If the syntax check fails, the reason is displayed and the program ends with exit code 1 (one).

```
 68 /**
 69  * Main - parse file name from command and call syntax check.
 70  */
 71 function main () {
 72     "use strict";
 73     // if wrong number of arguments, print usage message and exit
 74     if (process.argv.length !== 3) {
 75         console.log ("Usage: jsonsyntax file");
 76         console.log ("  file   JSON file to check syntax of");
 77         process.exit (1);
 78     }
 79
 80     // check syntax, displaying result message. Exit with success/fail code.
 81     try
 82     {
 83         var file = process.argv[2];
 84         checkSyntax (file);
 85         console.log ("File contains valid JSON content.");
 86         process.exit (0);
 87     } catch (e) {
 88         console.log ("Error: " + e.message);
 89         process.exit (1);
 90     }
 91 }
 92
 93 // if module invoked directly, call main
 94 if (require.main === module) {
 95     main ();
 96 }
```

The parser from the current *V8* runtime produces terse error messages that do not contain context information (such as line, position, or element the error occurred at). Thus, this tool is useful for pass / fail checking, but other tools will be more useful for editing activities.

Since a pass/fail exit code is returned, the program can be used in scripts. For example, a script can automate scanning all the *JSON* files in a directory to verify they all contain valid syntax.

To see a successful result, use the following command in the *chapter2* directory.

```
jsonsyntax array.json
```

To see an unsuccessful result, the *invalid.json* file is provided. It omits a comma after the *"address"* field in the first element, and uses an equals sign instead of a colon in the *"address"* field in the second element. Use the following command in the *chapter2* directory.

```
jsonsyntax invalid.json
```

If you have both the *Javascript* / *Node.js* and *Python* versions of the tool installed, you can use *jsonsyntaxn* instead of *jsonsyntax* as the command to run the *Javascript* / *Node.js* version explicitly.

Syntax Check Tool – Python Version

The syntax checking program in *Python* provides the same capability as the *Javascript* version, but the *Python* parser also provides location information for the error (e.g., "Expecting ',' delimiter: line 5 column 5 (char 67)").

The program logic is very similar to the *Javascript* / *Node.js* version, except the definition of the error used in exception handling in *Python* is a class derived from the built-in *Exception* class.

Directory: chapter2/python/jsonsyntax, file: jsonsyntax.py

The leading section includes the *Python* modules used (*sys*, *os.path* and *json*).

```
"""
Read a file and determine if content is valid JSON syntax.

Usage: jsonsyntax file
   file   JSON file to check syntax of

Exit code
  exit code: 0 for success, 1 for failure.
"""

# import modules
import sys
import os.path
import json
```

The class *CheckSyntaxError* is for exceptions. The passed parameters are stored for use by the exception handling code in the calling function.

```
class CheckSyntaxError (Exception):
    """
    Error definition thrown when an error occurs.
    - error code 1: Invalid name
    - error code 2: File does not exist
    - error code 3: Error reading file
    - error code 4: JSON syntax error

    Args:
        code: Error number
        message: Text message, suitable for display
    """
    def __init__ (self, code, message):
        self.code = code;
        self.message = message;

def checkSyntax (file):
    """
    Check syntax of file passed in command line argument.

    Args:
```

The *checkSyntax* function validates the input parameter, reads the specified file, and then parses the data from the file using the *JSON* parser. All errors are handled using exceptions, so if the function reaches its end, then the syntax check is successful.

```

    Raises:
        CheckSyntaxError
    """
    # verify file provided
    if (file == None):
        raise CheckSyntaxError (1, "Invalid name")

    # verify file exists
    if (os.path.isfile (file) == False):
        raise CheckSyntaxError (2, "File not found")

    # read specified file
    try:
        data = open (file).read ()
    except IOError as e:
        raise CheckSyntaxError (3, "Error reading file: " + e.strerror)
```

```
55
56      # parse the data as JSON
57      try:
58          json.loads (data)
59      except ValueError as e:
60          raise CheckSyntaxError (4, str (e.args));
61
62  def main ():
63      """
64      Main - parse file name from command and call syntax check.
```

The main function is executed if the program is called from the command line or in a script. It displays a usage message if the command does not include exactly one argument (*sys.argv* includes 1 static argument, so one program argument brings its length to 2). If the syntax check is successful, it displays a valid message and the program ends with exit code 0 (zero). If the syntax check fails, the reason is displayed and the program ends with exit code 1 (one).

```
66      # if wrong number of arguments, print usage message and exit
67      if (len (sys.argv) != 2):
68          print ("Usage: jsonsyntax fileName")
69          print ("  fileName   JSON file to check syntax of")
70          sys.exit (1)
71
72      # check syntax, displaying result message. Exit with success/fail code.
73      try:
74          # assign command line argument to file name
75          file = sys.argv[1]
76          checkSyntax (file);
77          print ("File contains valid JSON content.")
78          sys.exit (0);
79      except CheckSyntaxError as e:
80          print ("Error: " + e.message);
81          sys.exit (1);
82
83  if (__name__ == "__main__"):
84      main ()
```

To see a successful syntax check, use the following command in the *chapter* 2 directory.

```
jsonsyntax array.json
```

To see an unsuccessful result, the *invalid.json* file is provided. It omits a comma after the *"address"* field in the first element, and uses an equals sign instead of a colon in the *"address"* field in the second element. Use the following command in the *chapter2* directory.

```
jsonsyntax invalid.json
```

If you have both the *Javascript / Node.js* and *Python* versions of the tool installed, you can use *jsonsyntaxp* instead of *jsonsyntax* as the command to run the *Python* version explicitly.

3. JSON Schema

JSON Schema provides a content definition language for *JSON* file content. A *JSON Schema* definition is used to validate the structure and many aspects of the content of a *JSON* file.

A schema defined using *JSON Schema* uses the *JSON* syntax, making the previous chapter also the beginning of the introduction to creating *JSON Schema* definitions.

The *JSON Schema* specifications have been made available through the *IETF* (*Internet Engineering Task Force*) as *Internet Drafts*. The latest version of the core specification draft at the time of this writing, and which is used for the schema files included, is draft 4. There are three specifications related to *JSON Schema*,

The core specification draft

http://tools.ietf.org/html/draft-zyp-json-schema-04

The validation specification draft

http://tools.ietf.org/html/draft-fge-json-schema-validation-00

The hypertext specification draft

http://tools.ietf.org/html/draft-luff-json-hyper-schema-00

As *Internet Drafts*, they are not final specifications and when reading the specifications you may note that the expiry dates for some of the specifications have passed. This is part of the *IETF* process for publishing work in progress. While the work is continuing, the published materials are useful and there are tools available that conform to the continuing work that are available and of practical use.

When selecting tools and programs that implement *JSON Schema*, it is important to verify that the versions (drafts) the tool/program specifies compliance to is compatible with the version (draft) that your content has been created to.

Constructing a Schema

Schema creation can be a top-down or bottom-up activity. The top-down approach defines the schema first, and may precede the existence of actual data. The bottom-up approach uses actual data as the guide to create the schema. Especially for configuration files and simple datasets, the bottom-up approach is often more practical,

- These definitions tend to evolve as a project is defined or as the code is being written, rather than fixed in place ahead of time.
- Data content, such as enumerations of valid values, tend to be more common in message formats, configuration files, and specialized small datasets, than in general purpose databases.

The bottom-up approach also encourages active maintenance of the schema to reflect the needs of the content being managed, rather than treating the schema as a static resource.

To start working with *JSON Schema*, a simple empty schema will be presented first, followed by incrementally building more complex schemas.

Structure of a JSON Schema Definition

JSON Schema definitions, as *JSON* content, contain object, array, and name-value pair elements. Name-value pairs are defined for a the constructs *JSON Schema* defines, to provide schema processing elements for validation, and for describing the *JSON* content to be validated.

Schema processing elements include,

- *"$schema"* to specify the version of *JSON Schema* to process with.
- *"title"* and *"description"* to provide information about the schema to the reader.
- *"required"*, an array of elements, indicating which elements must be present.
- *"additionalProperties"*, often a boolean, to indicate whether existence of a property that is not specified is allowed (true) or not allowed (false). Alternatively, this can be a schema to constrain what additional properties are acceptable.
- *JSON* content definition elements are,
  - When defining a *JSON* object, *JSON Schema* uses the name-value pair *"type":"object"*. Elements contained within an object are defined under the name-value pair *"properties"*.
  - When defining a *JSON* array, *JSON Schema* uses the name-value pair *"type":"array"*. Elements contained within an object are defined under the name-value pair *"items"*.
  - When defining a *JSON* name-value pair, *JSON Schema* uses a name-value pair with the *"name"* of the element to match with the *JSON* content *"name"*, followed by an object that contains the constraints to be applied to the *JSON* element.

The next sections walk through the process of utilizing these schema elements.

Style

JSON Schema syntax does not impose positional requirements on elements. The position of braces {}, square brackets [], etc is up to the creator of the content.

The examples follow a style that favors readability in a page format, which includes keeping related content on the same line when possible. The guidelines for the style follow.

- Braces for objects and properties will occupy their own line, and left align with the object.
- Square brackets for arrays are inline.
- Constraints for a property or array will be listed on the same line.
- Indentation of 2 spaces for each level of indentation.
- When a property definition will not fit well on a single line, it will use the object definition style (braces on a new line, each constraint on a new line with an indent of two spaces).

There are many variations on style possible. Should you prefer a different style, development tools and "pretty parsers" can often reformat styles to suit.

Values for the "$schema" Element

The*"$schema"* element specifies the *JSON Schema* version that this content complies with. This allows the process reading/receiving the schema to know what content to expect per the version of the specification referenced.

The following shows a declaration without specifying a specific version. When used, the latest version of the specification known to the processing program will be used.

```
{
  "$schema":"http://json-schema.org/schema#"
}
```

If the schema uses features of a specific version, the version (draft presently) required can be specified in the URI. The following shows a declaration that specifies *draft-04* as the *JSON Schema* specification version to apply.

```
{
  "$schema":"http://json-schema.org/draft-04/schema#"
}
```

The list of valid values for this URI are specified in the core specification. For *draft 4* of the specification it is in section 6 titled *The "$schema" keyword*.

The examples provided in this book will use the *draft-04 URI*, since the features introduced in *draft-04* of the specification are used. Since the specification is expected to have additional versions, specifying the specific draft rather than using the latest version *URI* is recommended to ensure proper processing of schemas.

JSON Schema Empty Structure

To start, an empty schema will be defined. This is a very simple template for any *JSON Schema* definition.

Directory: chapter3, file: empty_schema.json.

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"schema name here",
  "description":"longer description here"
}
```

The first thing to note is that this is a valid *JSON* file. The enclosing braces { } indicate that a *JSON* object is being defined.

The first key-value pair, *"$schema"*, specifies the *JSON Schema* version that this content complies with. Per the introduction of the *"$schema"* element, the *draft-04 URI* is used.

The second and third key-value pairs are *"title"* and *"description"*. These are optional, but for the top level item it is recommended to provide context for the reader that may be more useful than the file name, *URL*, or other container name, for the schema.

The content specific to each schema definition will follow these leading elements.

JSON Schema Examples and Validation Tools

The remainder of this chapter provides hands on examples for different uses of *JSON Schema* and each of its features. The examples are incremental, starting with examples that show how different object and array constructs are defined. Structural constraints and content constraints follow, with use of references at the end.

For each example there are valid and invalid *JSON* files provided. With many validation use cases, understanding the invalid cases is often a better mechanism for learning. This is especially when the

invalid case enables a better understanding of how to utilize *JSON Schema* features to construct more precise schema definitions.

To perform the validation processing, a schema processor is used to determine whether the *JSON* content conforms to a schema definition. Three schema processors are provided, a command line tool written in *Javascript* using *Node.js*, a command line tool written in *Python*, and a web browser tool built using *HTML*, *CSS* and *Javascript*.

- The command line and web browser tools using *Javascript* use the schema processor in the *Tiny Validator for JSON Schema 4* library.
- The *Python* tool uses the schema processor in the *jsonschema* library.

The command line validation tool implementations are covered in chapter 8.

For each example, instructions for using the browser and command line options for validation processing. Valid and invalid cases are provided, along with an explanation for the results seen by each invalid case.

Using the Command Line Validation Tools

The command line validation tools accept the following parameters,

- Required. Name of the file containing the *JSON* content to be validated.
- Required. Name of the file containing the *JSON Schema* to be validated against.
- Optional: Name of individual files containing *JSON Schema* definitions that are accessed through references (*$ref*). Multiple files can be specified.
- Optional: Name of the file containing the *JSDB* contents. *JSDB* is a file containing one or more *JSON Schema* definitions, accessed through a *jsdb URI*. This is used in examples for custom resource managers.

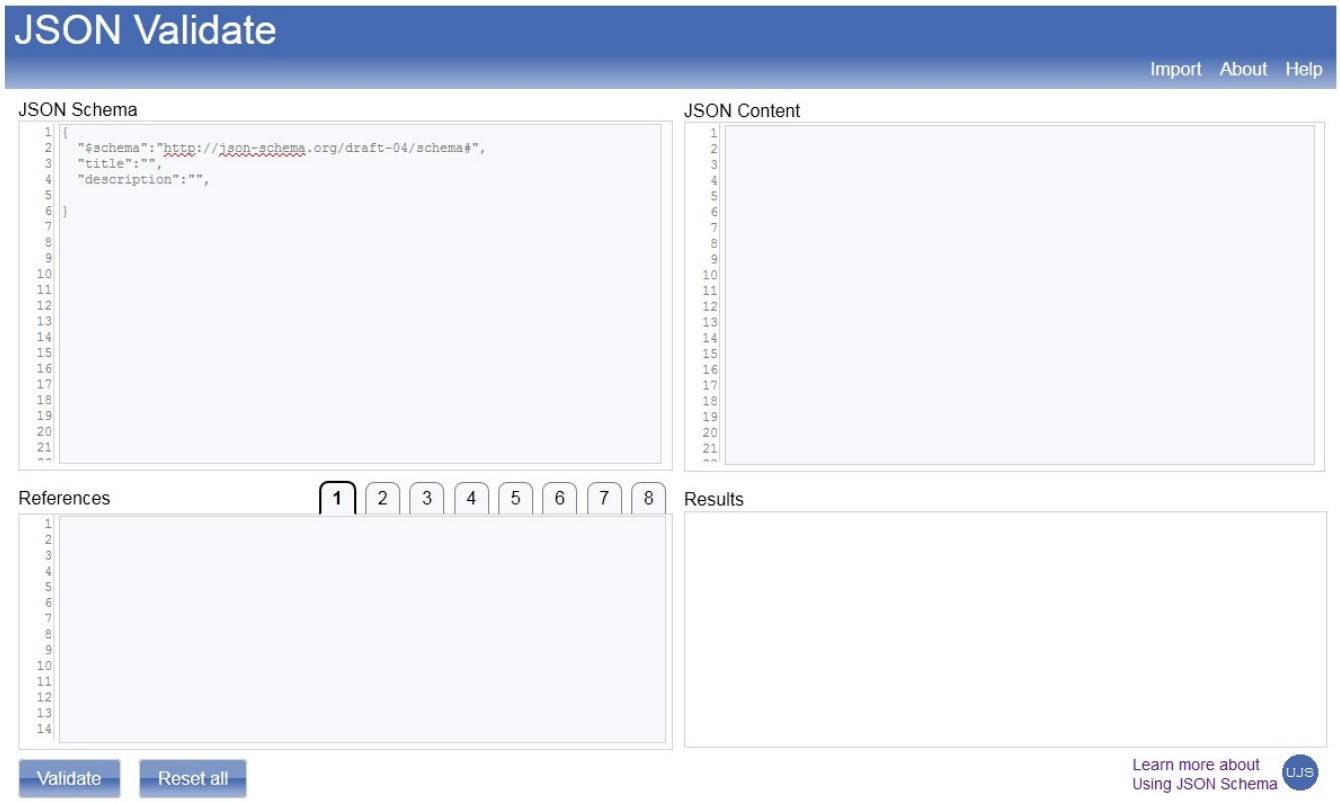
The installation process, covered in *Appendix A*, details the steps for installing the command line tools. After installation, the tools will be accessible from any directory on the computer. Each example includes the specific command for using the validation tool with the example.

Using the Online Validation Tool

An online validation tool is provided at,

http://jsonvalidate.com

It can be accessed using any current web browser. The following graphic is the *JSON Validate* web page as it is displayed when first loaded.



The *JSON Schema*, *JSON Content*, and *References* fields can be typed into directly, or content can be placed into the fields using *cut and paste*. As a simple tutorial, type or paste the following content into the *JSON Schema* field.

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Test",
  "description":"Test example",

  "type":"object",
  "properties":
  {
    "name":{"type":"string"}
  }
}
```

Then type or paste the following content into the *JSON Content* field.

```
{ "name":"John Doe" }
```

Press the *Validate* button to run the validation tool. The result will be display in the *Results* box at the bottom of the window. The result should be *Valid*. If text other than *Valid* is displayed, then check the schema and content text to make sure they match the example, and press *Validate* again after making corrections.

The *Reset All* button clears all the text areas, placing the boilerplate in the schema field.

In addition to using the tool for free form content, an import function is provided that provides easy access to all the book examples. To select an example from the book, click *Import* on the navigation bar. On the *Import* dialog, select the desired example and press the *Load* button. The schema, content and references for the example will be populated. You can experiment with the examples and use the *Validate* button at any time to run the validation processor.

Note: no changes are saved. To save any changes made, *cut and paste* the updated content to an editor (e.g., *Notepad* on *Windows* or *gedit* on *Linux* or T*extEdit* on *OS X*) and save the content using the editor save function.

JSON Schema for a Simple Object (3A)

The first example consists of a single object containing two properties.

Directory: chapter3, file: simpleObjectValid.json

```
{
  "address":"192.168.1.60",
  "port":80
}
```

The object itself does not express requirements or constraints on the content, so as part of defining the schema a set of decisions will be made that will be expressed in the schema. The basic schema definition starts by identifying the structure of the data.

Directory: chapter3, file:simpleObject_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Server",
  "description":"Simple IP server definition",

  "type":"object",
  "properties":
  {
    "address":{"type":"string"},
    "port":{"type":"integer"}
  }
}
```

Taking the original data content, the following process is applied in creating the initial schema.

- The original data content starts with an object definition (enclosing braces { }). Therefore, in the schema, the first element following *"title"* is the schema definition for an object (*"type":"object"*).
- The next content is the fields contained within the object. In *JSON Schema* these are defined as a set of properties. In the *JSON Schema* this is represented with the key-value pair *"properties"* with an element for each field that can be part of the data.
- For each property, a name-value pair is defined. The name portion is the name that is present in the data file (*"address"* and *"port"* in this example). The value portion is an object (enclosing braces { }) and always contains a key-value pair for the data type (*"type"="..."*). *"type"* is one of any, array, boolean, integer, number, *null*, object, or string.

This schema, when used for validating *JSON* content, will indicate as valid any *JSON* content that,

- Contains an object (enclosing braces { })
- If there is a key-value pair with the *"address"*, that the value associated with it is of type string.
- If there is a key-value pair with the name *"port"*, that the value associated with it is of type integer.

Looking at the list, you may have been expecting more definitive statements, such as verifying the *"address"* and *"port"* are present, since the schema definition seemed pretty prescriptive.

However, *JSON Schema* is permissive, it only validates against constraints that are explicitly defined. If a constraint is not specified for an element, then it is considered valid.

- If an element is not declared as *"required"*, then its constraints will only apply when the element is present in the content being validated. Otherwise, not being present is considered valid.
- If an unspecified property is present in the data content, assuming it was valid *JSON* syntax, the schema would accept it as valid if not explicitly constrained not to allow unspecified properties.

Validation (No Errors)

Running the validation against valid content is done with the provided validation tool using the following command in the chapter3 directory.

```
validate simpleObjectValid.json simpleObject_schema.json
```

Alternatively, the HTML tool can be used, loading the *Basic: Simple Object (Valid)* example.

The result of the validation is the valid message.

Validation (Invalid Structural Element)

The following JSON content places the object within an array element.

Directory: chapter3, file: simpleObjectInvalid1.json

```
[
  {
    "address":"192.168.1.60",
    "port":80
  }
]
```

To validate, use the *Basic: Simple Object (Invalid 1)* HTML example or use the command,

```
validate simpleObjectInvalid1.json simpleObject_schema.json
```

The schema specifies an object as the top level element, making the content invalid. The message conveys the expectation of an *object* element, but an *array* element being encountered instead.

Validation (Invalid Data Type)

Content with an incorrect data type will also be recognized as invalid.

Directory: chapter3, file: simpleObjectInvalid2.json

```
{
  "address":true,
  "port":80
}
```

To validate, use the *Basic: Simple Object (Invalid 2)* example or use the command,

```
validate simpleObjectInvalid2.json simpleObject_schema.json
```

The *boolean* value *true* is not acceptable for the *"address"* element, which the schema specifies as *string*. The error message will indicate the type of the value encountered does not match the type of the value expected.

Validation (Invalid Numeric Value)

Data types can have constraints that are part of their type definition. The following is an example that does not conform to a data type constraint.

Directory: chapter3, file: simpleObjectInvalid3.json

```
{
  "address":"192.168.1.60",
  "port":80.1
}
```

To validate, use the *Basic: Simple Object (Invalid 3)* example or use the command,

```
validate simpleObjectInvalid3.json simpleObject_schema.json
```

In the schema, the *port* element is defined as an *integer*. In the content presented for validation, the *port* element is defined with a floating point number. The invalid message will convey the mismatch in numeric representation.

Content Constraints for the Simple Object (3B)

To make the simple object schema definition more prescriptive, some constraints will be added. The content will require the properties defined to be present, and to not allow any properties other than those specified.

Directory: chapter3, file: simpleObjectReqd_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Server",
  "description":"IP server with required properties",

  "type":"object",
  "properties":
  {
    "address":{"type":"string"},
    "port":{"type":"integer"}
  },
  "additionalProperties":false,
  "required":["address", "port"]
}
```

At the same level as the *"properties"*, two additional constraints are defined.

- *"additionalProperties"* can be either a boolean value or a schema.
  - A boolean value of *true* will allow any additional properties to be included in the *JSON* content, and considered valid.
  - A boolean value of *false* will disallow any additional properties. If a property is defined that does not match a specified property, then the validation will fail.
  - A schema value will validate any properties not explicitly defined against this schema, determining whether the additional property is accepted or not.
- *"required"* is a key-value pair that is represented by an array containing a list of the properties that need to be present for the content to be considered valid.

With the addition of these constraints, the validation will only pass for *JSON* content that matches the structure specified in the schema with no extra content.

Validation (No Errors)

To validate against the original *JSON* content with the additional constraints, use the *Basic: Simple Object Required (Valid)* example or use the command.

```
validate simpleObjectValid.json simpleObjectReqd_schema.json
```

The result of the validation is the valid message.

Validation (Missing Property)

The following *JSON* content does not specify the *address* property.

Directory: chapter3, file: simpleObjectReqdInvalid1.json

```
{
  "port":80
}
```

To validate, use the *Basic: Simple Object Required (Invalid 1)* example or use the command,

```
validate simpleObjectReqdInvalid1.json simpleObjectReqd_schema.json
```

The missing property will be indicated in the error message displayed by the validation program.

Validation (Extraneous Property)

The following *JSON* content includes an extra property, *"name"*, that is not specified in the schema.

Directory: chapter3, file: simpleObjectReqdInvalid2.json

```
{
  "name":"Server 14",
  "address":"192.168.1.100",
  "port":80
}
```

To validate, use the *Basic: Simple Object Required (Invalid 2)* example or use the command,

```
validate simpleObjectReqdInvalid2.json simpleObjectReqd_schema.json
```

The extraneous property will be indicated in the error message displayed by the validation program.

JSON Schema for an Array (3C)

Using very simple *JSON* content for an array, a *JSON Schema* will be defined.

Directory: chapter3, file: simpleArrayValid.json

```
["Red", "Yellow", "Green"]
```

The schema will constrain the items in the array to be of type *string* only by including the *additionalItems* constraint set to *false*.

Directory: chapter3, file: simpleArray_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Traffic signals",
  "description":"List of traffic signal colors",

  "type":"array",
  "items":
  {
    "type":"string"
  },
  "additionalItems":false
}
```

Validation (No Errors)

To validate, use the *Basic: Simple Array (Valid)* example or use the command.

```
validate simpleArrayValid.json simpleArray_schema.json
```

The result of the validation is the valid message.

Validation (Extraneous Item)

The schema disallows additional items (*"additionalItems":false*), which means that all items in the content must conform to the schema definitions under *"items"*. Adding an array item that is does not conform will result in a validation error.

Directory: chapter3, file: simpleArrayInvalid.json

```
["Red", "Yellow", "Green", true]
```

To validate, use the *Basic: Simple Array (Invalid)* example or use the command,

```
validate simpleArrayInvalid.json simpleArray_schema.json
```

The *boolean* item will be indicated as invalid.

JSON Schema for a Named Object (3D)

From the preceding object and array examples, the definitions for the structural elements were shown for a single hierarchy level in the *JSON* content. As hierarchy levels are added, these schema reflects the structural hierarchy of the content. The following example shows a named object, introducing a hierarchy to the simple object example content.

Directory: chapter3, file: simpleNamedObjectValid.json

```
{
  "server":
  {
    "address":"192.168.1.60",
    "port":80
  }
}
```

The schema reflects the addition of the hierarchical element.

Directory: chapter3, file: simpleNamedObject_schema.json

```
 1 {
 2   "$schema":"http://json-schema.org/draft-04/schema#",
 3   "title":"Server",
 4   "description":"Server address and port number",
 5
 6   "type":"object",
 7   "properties":
 8   {
 9     "server":
10     {
11       "type":"object",
12       "properties":
13       {
14         "address":{ "type":"string" },
15         "port":{ "type":"integer" }
16       },
17       "additionalProperties":false,
18       "required":["address", "port"]
19     }
20   }
21 }
```

When comparing *simpleObjectReqd_schema.json* to this schema, we see,

- The definition of the innermost *"type":"object"* (lines 11-18) is the same. The only difference to this part of the definition is the hierarchical position it is defined within.
- At the top level, the first element *"type":"object"* (line 6) still represents the opening brace in the content.

- An additional hierarchy level is added at lines 7-9, where the *"server"* element is added.

This example establishes the structural model that will be applied. While the complexity of the content and hierarchical depth will vary, the structure of the schema follows this pattern.

Validation (No Errors)

To validate, use the B*asic: Simple Named Object (Valid)* example, or use the command.

```
validate simpleNamedObjectValid.json simpleNamedObject_schema.json
```

The result of the validation is the valid message.

JSON Schema for Mixed Objects and Arrays (3E)

The next configuration file contains a server definition (single object) and web page definition (array). The array contains a set of objects, but it is not required that the objects be named.

Directory: chapter3, file: simpleMixedValid.json

```
{
  "server":
  {
    "address":"192.168.1.60",
    "port":80
  },
  "pages":
  [
    {
      "url":"/",
      "page":"public/home.html"
    },
    {
      "url":"/about",
      "page":"public/company/about.html"
    },
    {
      "url":"/careers",
      "page":"public/company/careers.html"
    }
  ]
}
```

The schema adds an array type to the hierarchy, and a second named element under the top level. When the *JSON* content is being evaluated, the names are used to select the correct schema elements to apply to the content.

Directory: chapter3, file: simpleMixed_schema.json

```
 1 {
 2   "$schema":"http://json-schema.org/draft-04/schema#",
 3   "title":"HTTP Server",
 4   "description":"HTTP server definition and URL list",
 5
 6   "type":"object",
 7   "properties":
 8   {
 9     "server":
10     {
11       "type":"object",
12       "properties":
13       {
14         "address":{"type":"string"},
15         "port":{"type":"integer"}
```

```
16        },
17        "additionalProperties":false,
18        "required":["address", "port"]
19      },
20      "pages":
21      {
22        "type":"array",
23        "items":
24        {
25          "type":"object",
26          "properties":
27          {
28            "url":{"type":"string"},
29            "page":{"type":"string"}
30          },
31          "additionalProperties":false,
32          "required":["url", "page"]
33        }
34      }
35    }
36 }
```

The definitions of the properties that make up the individual elements (lines 14-15 and 28-29), along with the element constraints (lines 17-18 and 31-32) are packaged within the hierarchical elements (lines 9-19 and 20-34). The styles and characteristics are the same for both *object* and *array* types.

Validation (No Errors)

To validate, use Basic: Simple Mixed (Valid) example, or the following command in the chapter3 directory.

```
validate simpleMixedValid.json simpleMixed_schema.json
```

The result of the validation is the valid message.

This example completes the basic definition and structural definition aspects of *JSON Schema*. The next topics cover capabilities that allow crafting schema definitions that can reflect the complexity and nuances of the data being modeled for.

Patterns for Properties (3F)

The preceding examples have all used plain text for the property elements. When the validation of *JSON* content is performed, the matching of *JSON* content to schema elements will correlate the exact text for each.

However, *JSON Schema* also supports *patternProperties*, which allow the definition of a property to be a regular expression, rather than plain text. The validation process will then match the text definition from the *JSON* content using pattern matching against the patterns and plain text properties presented in the schema definition. For example, the following schema definition for a note uses a *patternProperties* element for comments.

Directory: chapter3, file: note_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Note",
  "description":"Note with optional comments",

  "type":"object",
  "properties":
```

```
    {
      "author":{"type":"string"},
      "title":{"type":"string"},
      "content":{"type":"string"}
    },
    "patternProperties":
    {
      "^comment[1-9]$":{"type":"string"}
    },
    "additionalProperties":false,
    "required":["author", "title", "content"]
  }
```

The patternProperties element contains the comments element defined as a regular expression. In this case, the allowed names for the elements in the *JSON content are comment1, comment2, comment3, comment4, comment5, comment6, comment7, comment8, and comment9. The following example JSON content shows an example note.*

Directory: chapter3, file: noteValid.json

```
{
  "author":"Jane",
  "title":"Party plan",
  "content":"Surprise party for Jim. Attendees: Sally, Bob so far.",
  "comment1":"Jill checking with Larry",
  "comment3":"Mary has to check schedule, will text later"
}
```

The *comment* properties can be any combination matching the pattern. In this example, two *comment* properties are defined, but they are not sequential.

Regular expressions are very flexible, and care should be taken to ensure patterns are scoped properly. The use of *carat (^)* and *dollar sign ($)* to indicate explicit start and end can ensure appropriate length of content for predictable parsing by the receiving program. For instance, the following shows the *patternProperties* definition without the explicit markers.

```
    "patternProperties":
    {
      "comment[1-9]":{ "type":"string" }
    },
```

This would allow "*comment1abc*", "*mycomment1*", "*commentcomment2*", et cetera to be used as property names. For the schema definition in this example, where the comment properties are not simply free form content, this arbitrary naming of properties would not be appropriate.

Validation (No Errors)

To validate, use the Pattern: Note (Valid) example, or use the following command in the *chapter3* directory.

```
validate noteValid.json note_schema.json
```

The result of the validation is the valid message.

Validation (Unmatched Pattern)

Showing a property that does not match the pattern, a property with the name *comment10* is used, which does not correspond with the defined pattern.

Directory: chapter3, file: noteInvalid.json

```
{
  "author":"Jane",
```

```
    "title":"Party plan",
    "content":"Surprise party for Jim. Attendees: Sally, Bob so far.",
    "comment1":"Jill checking with Larry",
    "comment10":"Mary has to check schedule, will text later"
  }
```

To see the validation message, use the *Pattern: Note (Invalid)* example, or use the command,

```
validate noteInvalid.json note_schema.json
```

The validation program will show that the *comment10* element is invalid. Note that *additionalProperties* must be set to *false* in the schema definition, otherwise the *comment10* element will be accepted – though it will not be subject to the *patternProperties* definition, as it will be treated the same as any other undefined property.

Dependencies for Properties (3G)

A last structural element is *"dependencies"*. A dependency allows the presence of one or more properties to be conditional on the presence of a specified property. This provides a more flexible definition than afforded by the *"required"* element for schema definitions that have groups of properties related to each other.

Consider the content for online orders from a company that has a loyalty program. A couple of orders are shown in the following *JSON* content.

Directory: chapter3, file: orderValid.json

```
{
  "orders":
  [
    {
      "order":"123456789",
      "billTo":"Jane Doe",
      "billAddress":"1234 Elm St, Anytown PA 12345",
      "shipTo":"John Public",
      "shipAddress":"2345 Oak St, Anytown PA 12346",
      "loyaltyId":"A123456",
      "loyaltyBonus":"Free shipping"
    },
    {
      "order":"234567890",
      "billTo":"Jack Smith",
      "billAddress":"111 Main St, Anytown PA 12345"
    }
  ]
}
```

For each order, the billing and shipping information may be the same (e.g., self purchase) or different (e.g., purchase for someone else). Participation in a loyalty program is optional, but if participating, then there may be a benefit associated with an order. The schema for the order is shown next.

Directory: chapter3, file: order_schema.json

In the schema definition, there are three groups of properties,

- Lines 16-18 are the common properties that are in every order. They are included in line 25 as the *"required"* properties.
- Lines 19-20 are the shipping properties. These are optional as a group, since the billing information will be used if these are not present. However, if the *"shipTo"* property is defined, then the *"shipAddress"* must also be defined. This dependency is defined on line 28.

- Lines 21-22 are the properties populated when the person making the order is part of the loyalty program, otherwise neither of the properties in this group will be present. If a *"loyaltyId"* is present, then the *"loyaltyBonus"* to apply to this order must also be present. This dependency is defined on line 29.

```
 1 {
 2   "$schema":"http://json-schema.org/draft-04/schema#",
 3   "title":"Order",
 4   "description":"Order billing and shipping information",
 5
 6   "type":"object",
 7   "properties":
 8   {
 9     "orders":
10     {
11       "type":"array",
12       "items":
13       {
14         "properties":
15         {
16           "order":{"type":"string"},
17           "billTo":{"type":"string"},
18           "billAddress":{"type":"string"},
19           "shipTo":{"type":"string"},
20           "shipAddress":{"type":"string"},
21           "loyaltyId":{"type":"string"},
22           "loyaltyBonus":{"type":"string"}
23         },
24         "additionalProperties":false,
25         "required":["order", "billTo", "billAddress"],
26         "dependencies":
27         {
28           "shipTo":["shipAddress"],
29           "loyaltyId":["loyaltyBonus"]
30         }
31       }
32     }
33   }
34 }
```

The use of dependencies provides an easy to understand mechanism for defining relationships between properties. For more complex relationships between properties, the *oneOf*, *anyOf*, or *allOf* mechanisms can be used.

Validation (No Errors)

To validate, use the Dependency: Order (Valid) example, or use the following command in the chapter3 directory.

```
validate orderValid.json order_schema.json
```

The result of the validation is the valid message.

Validation (Missing Dependency)

When a property listed as a dependency is missing, the validation will fail. The following order is missing the shipping address for an order that has a person specified to ship the order to.

Directory: chapter3, file: orderInvalid.json

```
{
  "orders":
  [
```

```
    {
      "order":"123456789",
      "billTo":"Jane Doe",
      "billAddress":"1234 Elm St, Anytown PA 12345",
      "shipTo":"John Public",
      "loyaltyId":"A123456",
      "loyaltyBonus":"Free shipping"
    }
  ]
}
```

To see the validation message, use the *Dependency: Order (Invalid)* example, or the command,

```
validate orderInvalid.json order_schema.json
```

The message will indicate that the element required to fulfill the dependency (*shipAddress*) is not present.

Choices: allOf, oneOf, anyOf (3H)

The *requires* and *dependencies* constraints provide fairly coarse grained mechanisms for conveying what elements need to be present to the validation processor. When finer grain control is desired, the *allOf*, *oneOf*, and *anyOf* mechanisms can be used. Each of these elements contain an array, with each element of the array representing content that will be matched against. The choice of *allOf*, *oneOf* or *anyOf* determines how the validation processor will treat the results of the matches,

- *allOf* requires that all elements in the array are matched successfully.
- *oneOf* requires one, and only one, of the elements in the array to match successfully.
- *anyOf* requires one or more of the elements in the array to be matched successfully.

Schema definitions can use *allOf*, *oneOf*, and *anyOf* individually or in combination, providing significant flexibility for defining elements that have complex definitions or contextual relationships.

These constraints can apply to values or to properties. When applied to values, the constraint is defined inline. When applied to properties, the constraint is defined in a manner similar to the required and additionalProperties constraints. In the latter case, the constraint will have the appearance of an independent element to the reader of the schema. The examples in this chapter focus on examples where values are constrained. *Chapter 4* provides examples where properties are constrained, with more detail on schema design for these cases.

AllOf

The default processing for validation acts as if the *allOf* constraint is present when doing content validation. The *requires* and *additionalProperties* constraints provide efficient mechanisms for ensuring the presence of the desired structural elements. Between these, the use of *allOf* will typically be for improving conciseness in more complex schema definitions. The example below shows two representations of the same definition, the first using *allOf*, and the second without.

Directory: chapter3, file: choiceAllOf_schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title":"Registration",
  "description":"Sports activity registration, 12-14 age bracket",

  "type":"object",
  "properties":
```

```
    {
      "name":{"type":"string"},
      "sex":{"allOf":[{"type":"string"}, {"enum":["M", "F"]}]},
      "age":{"allOf":[{"type":"integer"}, {"minimum":12}, {"maximum":14}]}
    },
    "additionalProperties":false,
    "required":["name", "sex", "age"]
  }
```

File: choiceAllOfImplied_schema.json.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title":"Registration",
  "description":"Sports activity registration, 12-14 age bracket",

  "type":"object",
  "properties":
  {
    "name":{"type":"string"},
    "sex":{"type":"string", "enum":["M", "F"]},
    "age":{"type":"integer", "minimum":12, "maximum":14}
  },
  "additionalProperties":false,
  "required":["name", "sex", "age"]
}
```

As seen in the definitions for the content constraints for the properties *"sex"* and *"age"*, the use of *allOf* in the first example explicitly specifies the matching criteria. In the second example, the matching criteria is implied, as the default processing of the validation will enforce matching against all specified criteria items.

Validation (No Errors)

The following *JSON* content will validate with either of these schema definitions.

Directory: chapter3, file: choiceAllOfValid.json

```
{
  "name":"John Doe",
  "sex":"M",
  "age":12
}
```

To validate using the explicitly defined schema, use the *Choice: All Of (Valid)* example, or use the following command in the *chapter3* directory.

```
validate choiceAllOfValid.json choiceAllOf_schema.json
```

To validate using the implicitly defined schema, use the *Choice: All Of Implied (Valid)* example, or use the following command in the *chapter3* directory.

```
validate choiceAllOfValid.json choiceAllOfImplied_schema.json
```

OneOf

When an exclusive choice between elements is desired, the *oneOf* definition permits an array of choices to be defined from which one, and only one, must be present.

The national portion of an international address is defined per country. In the following example, the national portion of the address is shown for Canada, the United States of America, and Mexico.

Note, for this example, only three states/provinces are shown for each country, rather than the full list.

Directory: chapter3, file: choiceOneOf_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"National address",
  "description":"National portion of an address",

  "type":"object",
  "oneOf":
  [
    {
      "properties":
      {
        "country":{"type":"string", "enum":["CAN"]},
        "province":{"type":"string", "enum":["AB", "BC", "MB"]},
        "postalCode":{"type": "string",
            "pattern":"^[A-Z][0-9][A-Z][0-9][A-Z][0-9]$"}
      },
      "additionalProperties":false,
      "required":["country", "province", "postalCode"]
    },
    {
      "properties":
      {
        "country":{"type":"string", "enum":["USA"]},
        "state":{"type":"string", "enum":["AL", "AK", "AR"]},
        "zipCode":{"type":"string", "pattern":"^[0-9]{5}(-[0-9]{4})?$"}
      },
      "additionalProperties":false,
      "required":["country", "state", "zipCode"]
    },
    {
      "properties":
      {
        "country":{"type":"string", "enum":["MEX"]},
        "state":{"type":"string", "enum":["AGS", "BC", "BCS"]},
        "postalCode":{"type": "string", "pattern":"^[0-9]{5}$"}
      },
      "additionalProperties":false,
      "required":["country", "state", "postalCode"]
    }
  ]
}
```

The validation processor will determine how many choices the content presented matches. If the number of matches is exactly one, then the content will be accepted.

Validation (No Errors)

The following shows a valid address to be processed against the schema.

Directory: chapter3, file: choiceOneOfValid.json

```
{
  "country":"USA",
  "state":"AK",
  "zipCode":"99501"
}
```

To validate, use the *Choice: One Of (Valid)* example, or use the following command.

```
validate choiceOneOfValid.json choiceOneOf_schema.json
```

The content matches the USA address definition for the *country* property enumeration, use of the property name *state*, valid enumeration for the property *state*, use of the property name *zipCode*, and format of the *zipCode*.

The content does not match either of the Canada or Mexico definitions, thus the content is valid.

Validation (No Match)

In this address, the value of the *country* property is incorrect for the remainder of the content.

Directory: chapter3, file: choiceOneOfInvalid.json

```
{
  "country":"USA",
  "province":"AB",
  "postalCode":"A1B2C3"
}
```

To see the validation message, use the *Choice: One Of (Invalid)* example, or the command,

```
validate choiceOneOfInvalid.json choiceOneOf_schema.json
```

The validation processor will find the following as it matches against each choice,

- The *USA* definition will match on *country*, but have incorrect property names for the other elements.
- The *CAN* definition will not match the *country* enumeration, but will match the other properties.
- The *MEX* definition will not match the *country* enumeration, state property, or the format for *postalCode*.

Since there is no matching schema in the selection of *oneOf* choices, the validation fails.

AnyOf

When using anyOf condition, a property can be validated a set of criteria that may include criteria that can match multiple times.

In this example, a play day activity is being set up for a school that has many different class sizes. Four team sizes are defined, and for some team sizes, the team can select which category they want to compete in. A team size of 4 or 10 provides this choice of category.

Directory: chapter3, file: choiceAnyOf_schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title":"Team size",
  "description":"Team sizes for play day competitions",

  "type":"object",
  "properties":
  {
    "size":
    {
      "anyOf":
      [
        {"type":"integer", "minimum":2, "maximum":4},
        {"type":"integer", "minimum":4, "maximum":6},
        {"type":"integer", "minimum":8, "maximum":10},
        {"type":"integer", "minimum":10, "maximum":12}
      ]
    }
  },
```

```
    "additionalProperties":false,
    "required":["size"]
  }
```

The validation processor tries to match the content against each of the choices defined within the *anyOf* array. If a match is found for any element, including matching multiple elements, then the content will be accepted.

Validation (No Errors)

A team size of 4 will match two of the criteria of the *anyOf* choices.

Directory: chapter3, file: choiceAnyOfValid.json

```
{
  "size":4
}
```

To validate, use the *Choice: Any Of (Valid)* example, or use the following command.

```
validate choiceAnyOfValid.json choiceAnyOf_schema.json
```

The team size of 4 matches the first criteria (2 – 4) and second criteria (4-6), but not the third (8-10) or fourth (10-12). The valid content message is displayed.

Validation (No Match)

A team size of 7 will not match any of the *anyOf* choices.

Directory: chapter3, file: choiceAnyOfInvalid.json

```
{
  "size":7
}
```

To see the validation message, use the *Choice: Any Of (Invalid)* example, or the command,

```
validate choiceAnyOfInvalid.json choiceAnyOf_schema.json
```

The validation processor will try to match against each choice, without success. The message will indicate the lack of a matching choice.

The Negative Constraint: not

Sometimes the desired validation behavior is to model a negative constraint, that is, to resolve an element as valid only when it fails to meet validation criteria.

In the following example, registration for a sports team is being collected including the name, age and league that the person is signing up for. This registration does not include signing up for the intramural league, which is handled by a separate registration process. The schema therefore uses a negative constraint for the *"league"* property, excluding the choice of "*intramural"* as a valid value for this property.

Directory: chapter3, file: register_schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title":"Team registration",
  "description":"Team registration, non-intramural leagues only",

  "type":"object",
  "properties":
  {
    "name":{"type":"string"},
    "age": {"type":"integer", "minimum":10, "maximum":14},
```

```
      "league": {"type":"string", "not":{"enum":["intramural"]}}
    }
  }
```

Negative constraints can be sometimes be difficult to interpret, so information in the description of the schema can be helpful to understanding the context of any negative constraints.

Validation (No Errors)

This example uses a registration that contains an allowed value for the *"league"* property.

Directory: chapter3, file: registerValid.json

```
{
  "name":"John Doe",
  "age":12,
  "league":"District A"
}
```

To validate, use the *Not: Register (Valid)* example, or use the following command.

```
validate registerValid.json register_schema.json
```

The *"league"* value is not *"intramural"*, which fulfills the not constraint. The valid content message is displayed.

Validation (Invalid Value)

The *"not"* constraint reverses the match result, as shown in the following example.

Directory: chapter3, file registerInvalid.json

```
{
  "name":"John Doe",
  "age":12,
  "league":"intramural"
}
```

To see the validation message, use the *Not: Register (Invalid)* example, or the command,

```
validate registerInvalid.json register_schema.json
```

The validation processor will recognize first the match of the value *"intramural"* with the enumeration specified for the *"league"* property. The processor will then apply the *"not"* constraint, changing the validation state from a valid state to an invalid state. The message from the validation processor will indicate that the content did not comply with the *"not"* condition for the property.

Object and Array Constraints (3I)

Object and array constraints allow the schema to apply constraints to object (property) and array (item) definitions. Constraints include number, naming and existence requirements. Cross element constraints are also available through the choice constraints (*allOf*, *anyOf*, *oneOf*, and *not*).

Object Constraints

In the preceding examples, a number of constraints on object definitions have been introduced. These include the *additionalProperties* and *required* constraints.

In addition, the number of properties allowed can be constrained, with *minProperties* and *maxProperties*. These specify the minimum and maximum number of properties (inclusive) that may be present. This constraint can be used to ensure that free form content is present (*minProperties*) and that a reasonable limit can be placed on the number of properties (*maxProperties*). These provide both

guidance to the creation of the content, while also allowing programs to anticipate the processing required for content received.

Address books allow multiple contact methods to be associated with a contact. This arbitrary set of methods and addresses can be defined in a schema, while limiting the total number allowed.

Directory: chapter3, file: contact_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Contacts",
  "description":"Contacts: name, freeform contact methods",

  "type":"object",
  "properties":
  {
    "name":{"type":"string"}
  },
  "minProperties":1,
  "maxProperties":5,
  "required":["name"]
}
```

Validation (No Errors)

Valid content includes the *"name"* property (since it is defined as a *"required"* property) and 0 to 4 additional properties.

Directory: chapter3, file: contactValid.json

```
{
  "name":"John",
  "mobile":"555 555-1212",
  "email":"john@example.com"
}
```

To validate, use the *Properties: Contact (Valid)* example, or use the following command.

```
validate contactValid.json contact_schema.json
```

The total number of properties is 3, which is consistent with the bounds set in the schema.

Validation (Too Many Properties)

This example satisfies the requirement that *"name"* is present. However, with 5 contact properties, the total number of properties is 6.

Directory: chapter3, file: contactInvalid.json

```
{
  "name":"John",
  "home":"555 555-1111",
  "mobile":"555 555-1212",
  "work":"555 555-2222",
  "email":"john@example.com",
  "twitter":"@John"
}
```

To validate, use the *Properties: Contact (Invalid)* example, or use the following command.

```
validate contactInvalid.json contact_schema.json
```

The validation fails with a message indicating the total number of properties (6) exceeds the constraint *"maxProperties"* set in the schema (5).

Custom additionalProperties Constraint

In addition to limiting the number of additional properties, the content of each can also be constrained. Instead of the boolean value *true*, the *additionalProperties* definition contains schema constraints. Extending the contacts example, in this case the additional contacts will be constrained to allow only North American phone numbers.

In the following schema, the additional property content is constrained to being a *string* and conforming to a North American phone number (such as 555 555-1212).

Directory: chapter3, file: contactPhone_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Contacts",
  "description":"Contacts: name, freeform contact methods",

  "type":"object",
  "properties":
  {
    "name":{"type":"string"}
  },
  "minProperties":1,
  "maxProperties":5,
  "required":["name"],
  "additionalProperties":
  {
    "type":"string", "pattern":"^([0-9]{3}) [0-9]{3}-[0-9]{4}$"
  }
}
```

Validation (No Errors)

By only including phone number contact information in the additional properties, the following schema is acceptable.

Directory: chapter3, file: contactPhoneValid.json

```
{
  "name":"John",
  "mobile":"555 555-1212",
  "email":"john@example.com"
}
```

To validate, use the *Properties: Contact Phone (Valid)* example, or use the following command.

```
validate contactPhoneValid.json contactPhone_schema.json
```

The regular expression is simplistic in this example, but can be more sophisticated.

Validation (Invalid Content)

In this content, an email address is included as one of the additional properties. Since the constraint does not allow any content other than North American phone numbers, the validation will fail.

Directory: chapter3, file: contactPhoneInvalid.json

```
{
  "name":"John",
  "home":"555 555-1111",
  "mobile":"555 555-1212",
  "work":"555 555-2222",
  "email":"john@example.com"
}
```

To validate, use the *Properties: Contact Phone (Invalid)* example, or use the following command.

```
validate contactPhoneInvalid.json contactPhone_schema.json
```

The validation fails with a message indicating content of the email property is invalid.

Array Constraints

Array definitions can be constrained to a range of items allowed using the *minItems* and *maxItems* constraints. In addition, the array can be constrained to only allow arrays that have no duplicate items using the *uniqueItems* constraint. These are very useful constraints to ensure programs are receiving an expected number of items, such as a primary server and backup server that are not at the same address and port number, as shown in this example.

Note that in this schema example, the *"minItems"* and *"maxItems"* constraints are on the same line as the *"type":"array"*. The *"uniqueItems"* and *"additionalItems"* constraint still follow the *"items"* element. This shows the freedom to arrange the content as desired, in this case to associate the range of allowed items with the array (which is a familiar construct in array definition syntax in many other technologies). The characteristics that apply to the elements themselves are then placed in the position following the item content definition. It is not incorrect syntax to place *"minItems"* and *"maxItems"* following the *"items"* element, equivalent to the prior example which placed *"minProperties"* and *"maxProperties"* in that manner.

Directory: chapter3, file:serverArray_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Primary and Backup Servers",
  "description":"IP address/port for primary/backup server pair",

  "type":"array", "minItems":2, "maxItems":2,
  "items":
  {
    "type":"object",
    "properties":
    {
      "address":{"type":"string"},
      "port":{"type":"integer", "minimum":0, "maximum":65535}
    },
    "required":["address", "port"]
  },
  "uniqueItems":true,
  "additionalItems":false
}
```

Arrays, like properties, can be constrained in acceptance of items not otherwise specified in the *"items"* definition. The boolean constraint *"additionalItems"* is used. A value of *false* indicates that only items matching the specified *"items"* list are acceptable.

Validation (No Errors)

The following example shows two server definitions, that do not share the same address / port definition.

Directory: chapter3, file: serverArrayValid.json

```
[
  {
    "address":"192.168.1.60",
    "port":80
  },
  {
    "address":"192.168.1.61",
```

```
    "port":80
  }
]
```

To validate, use the *Array: Server Array (Valid)* example, or use the following command.

```
validate contactValid.json contact_schema.json
```

The number of items in the array is 2, which fulfills the constraint defined in the schema.

Validation (Incorrect Number of Items)

In this example, only one server is defined, the backup server is missing.

Directory: chapter3, file: serverArrayInvalid1.json

```
[
  {
    "address":"192.168.1.60",
    "port":80
  }
]
```

To validate, use the *Array: Server Array (Invalid 1)* example, or use the following command.

```
validate serverArrayInvalid1.json serverArray_schema.json
```

The validation fails with a message indicating the number of items present does not meet the minimum number defined in the schema.

Validation (Duplicate Items)

In this example, the correct number of items is defined, but the items are identical.

Directory: chapter3, file: serverArrayInvalid2.json

```
[
  {
    "address":"192.168.1.60",
    "port":80
  },
  {
    "address":"192.168.1.60",
    "port":80
  }
]
```

To validate, use the *Array: Server Array (Invalid 2)* example, or use the following command.

```
validate serverArrayInvalid2.json serverArray_schema.json
```

The validation fails with a message indicating the items are not unique, as required by the schema.

Value Constraints (3J)

In the examples so far, elements have been defined with their type defined. This enables a fairly coarse grained validation of content. For example, an element defined as *"type":"integer"* can be checked to ensure is contains only numbers or the symbols plus or minus.

Additional constraints can be specified that provide finer grain validation. For example, number ranges can be added to a port number definition. In the following example, see the constraints that have been added to the *"port"* property.

Directory: chapter3, file: server_schema.json.

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
```

```
  "title":"Server",
  "description":"Server name and IP address, port",

  "type":"object",
  "properties":
  {
    "name":{"type":"string"},
    "address":{"type":"string"},
    "port":{"type":"integer", "minimum":0, "maximum":65535}
  },
  "additionalProperties":false,
  "required":["name", "address", "port"]
}
```

The addition of the constraints will generate validation messages when the value specified for the port number is a negative number or exceeds 65535.

Validation (Valid)

Directory: chapter3, file: serverValid.json

```
{
  "name":"Server 14",
  "address":"192.168.1.60",
  "port":80
}
```

To validate, use the *Range: Server (Valid)* example, or use the following command.

```
validate serverValid.json server_schema.json
```

The *"port"* value is in the range 0 to 65535, which fulfills the not constraint. The valid content message is displayed.

Validation (Value Out of Range)

Directory: chapter3, file: serverInvalid.json

```
{
  "name":"Server 14",
  "address":"192.168.1.60",
  "port":100000
}
```

To see the validation message, use the *Range: Server (Invalid)* example, or the command,

```
validate serverInvalid.json server_schema.json
```

In the invalid definition, the value of 100000 for the port number will be flagged as out of range for the "port" property, and the validation fails.

Integer and Number: Minimum/Maximum Constraints

Range constraints can be bounded on both ends or only one end. In the previous example, the range for *"port"* was constrained on both ends using minimum and maximum. The following example constrains only the minimum value, allowing any positive integer for the *"quantity"*.

```
"properties":
{
  "quantity":{"type":"integer", "minimum":1}
}
```

Integer and Number: Exclusive Minimum/Maximum Constraints

Range constraints can inclusive or exclusive, meaning that the boundary of the range is included or excluded. By default, the *exclusiveMinimum* and *exclusiveMaximum* constraints are false, making

minimum and maximum inclusive. For example, a scientific measurement may require a distance value that is non-zero, but accept values less than one. Using the *exclusiveMinimum* constraint allows this to be specified clearly without creating an arbitrary fraction for the minimum value.

Directory: chapter3, file: distance_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Distance",
  "description":"Distance, must be greater than zero.",

  "type":"object",
  "properties":
  {
    "distance":{"type":"number", "minimum":0.0, "exclusiveMinimum":true}
  }
}
```

Validation (No Errors)

The following content defining a small distance is valid.

Directory: chapter3, file: distanceValid.json

```
{
  "distance":0.01
}
```

To validate, use the *Number: Distance (Valid)* example, or using the following command.

```
validate distanceValid.json distance_schema.json
```

Validation (Equal to Exclusive Constraint)

However, defining a distance of 0.0 will be invalid as shown in the following example.

Directory: chapter3, file: distanceInvalid.json

```
{
  "distance":0.0
}
```

To validate, use the *Number: Distance (Invalid)* example, or using the following command.

```
validate distanceInvalid.json distance_schema.json
```

The validation message will indicate the content is not valid since the value provided is equal to the exclusive minimum constraint value.

Integer and Number: MultipleOf Constraint

A value constraint, *multipleOf*, specifies that a value must be a multiple of a specified number. For instance, if the *multipleOf* is specified as 4, then the values 8, 32, and 1024 would be valid, whereas 3, 19, and 1027 would not be valid. For example, to ensure a measurement provided in inches is always an even foot, a *multipleOf* value of 12 can be used.

In this example, a measurement is defined that will represent the length of wood as measured in inches. However, the constraint will require that the length be an even number of feet (that is, a multiple of 12).

Directory: chapter3, file: measure_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Measure",
  "description":"Inches measurement, must be even number of feet",
```

```
  "type":"object",
  "properties":
  {
    "length":{"type":"integer", "minimum":0, "multipleOf":12}
  }
}
```

Validation (No Errors)

The following content defining a length of 72 inches (6 feet, zero inches) is valid.

Directory: chapter3, file: measureValid.json

```
{
  "length":72
}
```

To validate, use the *Number: Measure (Valid)* example, or using the following command.

```
validate measureValid.json measure_schema.json
```

Validation (Not Multiple Of)

Changing the length to 74 inches (6 feet 2 inches).

Directory: chapter3, file: measureInvalid.json

```
{
  "length":74
}
```

To validate, use the *Number: Measure (Invalid)* example, or using the following command.

```
validate measureInvalid.json measure_schema.json
```

With the length no longer being a multiple of 12, the validation message will indicate the value does not meet the *multipleOf* constraint defined in the schema.

String: Length Constraints

Strings can be constrained in their length, both minimum and maximum. For example, first name, last name and middle name can be constrained for applications that need to provide printed materials that have predefined locations for name information that have limited space.

Directory: chapter3, file: name_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Name",
  "description":"Full name, middle name optional",

  "type":"object",
  "properties":
  {
    "firstName":{"type":"string", "minLength":1, "maxLength":10},
    "middleName":{"type":"string", "minLength":0, "maxLength":8},
    "lastName":{"type":"string", "minLength":1, "maxLength":20}
  },
  "required":["firstName","lastName"]
}
```

From this definition,

- The first name is required and must have between 1 and 10 characters.
- The middle name is optional, but when present can be empty or have no more than 8 characters.

- The last name is required and must have between 1 and 20 characters.

Validation (No Errors)

A name definition that is valid.

Directory: chapter3, file: nameValid.json

```
{
  "firstName":"John",
  "middleName":"David",
  "lastName":"Doe"
}
```

To validate, use the *String: Name (Valid)* example, or using the following command.

```
validate nameValid.json name_schema.json
```

Validation (Invalid String Length)

Providing a name that has a long middle name.

Directory: chapter3, file: nameInvalid.json

```
{
  "firstName":"Jane",
  "middleName":"Alexandria",
  "lastName":"Doe"
}
```

To validate, use the *String: Name (Invalid)* example, or using the following command.

```
validate nameInvalid.json name_schema.json
```

With the middle name exceeding 8 characters in length, the validation will indicate the property value exceeds the *"maxLength"* constraint.

String: Pattern Constraint

Strings may also be constrained to match a pattern. The pattern is defined using a regular expression, which is resolved according to the *ECMAScript* definition (*ECMA 262*). For example, a United States zip code can be specified using the following regular expression.

```
^[0-9]{5}(-[0-9]{4})?$
```

Which allows 5 digits followed by an optional hyphen and 4 digits. The carat (^) and dollar sign ($) indicate the zip code must be its own text entity. Using this format in a zip code property can enhance the ability of the validation processor to recognize not just that the value presented is a string, but also that it matches the pattern defined.

Directory: chapter3, file: zipCode_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"US zip code",
  "description":"US zip code with pattern to validate against",

  "type":"object",
  "properties":
  {
    "zipCode":{"type":"string", "pattern":"^[0-9]{5}(-[0-9]{4})?$"}
  }
}
```

Validation (No Errors)

A 5+4 zip code is defined.

Directory: chapter3, file: zipCodeValid.json

```
{
  "zipCode":"12345-6789"
}
```

To validate, use the *String: Zip Code (Valid)* example, or using the following command.

```
validate zipCodeValid.json zipCode_schema.json
```

Validation (No Pattern Match)

Using the same zip code numbers, but excluding the hyphen.

Directory: chapter3, file: zipCodeInvalid.json

```
{
  "zipCode":"123456789"
}
```

To validate, use the *String: Zip Code (Invalid)* example, or using the following command.

```
validate zipCodeInvalid.json zipCode_schema.json
```

When the pattern for the content does not match, the validation processor will indicate that the element being validated does not meet the pattern constraint.

String: Format Constraint

Some commonly used data type formats like date and IP addresses are defined in the specification. This is a convenience, as the equivalent function can also be achieved using *"pattern"* in data type definitions. The data types defined for *"format"* in *draft 4* of the specification are *"date-time"*, *"email"*, *"hostname"*, *"ipv4"*, *"ipv6"* and *"uri"*. For example,

```
"properties":
{
  "email":{ "type":"string", "format":"email" }
}
```

Currently, the support of *"format"* by validation processors is optional, so the examples in this book will use *"pattern"* instead, along with the regular expression for validating against these data types.

Enumerations

A lot of programming code is devoted to determining the correctness of values provided to the program. This reflects the variety of manners in which data can be sourced, especially for content that is entered by a person interactively or in a form that can be hand edited. However, other factors such as old data that has not been maintained or incorrect function in another application can also present invalid data to a program.

Enumerations provide a way to use validation to enforce correctness of data when the data item is constrained to a particular set of values. For instance, the value for the state in a United States postal address is one of a specific set of values.

Directory: chapter3, file: postUSA_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"US postal abbreviations",
  "description":"Postal abbreviations: US states/territories",

  "type":"object",
```

```
  "properties":
  {
    "state":
    {
      "type":"string",
      "enum":["AL", "AK", "AR", "AS", "AZ", "CA", "CO", "CT",
              "DC", "DE", "FL", "FM", "GA", "GU", "HI", "IA",
              "ID", "IL", "IN", "KS", "KY", "LA", "MA", "MD",
              "ME", "MH", "MI", "MN", "MO", "MP", "MS", "MT",
              "NC", "ND", "NE", "NH", "NJ", "NM", "NV", "OH",
              "OK", "OR", "PA", "PR", "PW", "RI", "SC", "SD",
              "TN", "TX", "UT", "VA", "VI", "VT", "WA", "WI",
              "WV", "WY", "AA", "AE", "AP"]
    }
  }
}
```

When an enumeration is defined, the validation processor will verify that the content being validated is of the correct type and that it matches one of the values listed within the array associated with the enumeration.

Validation (No Errors)

The state of Hawaii (HI) is presented in the example.

Directory: chapter3, file: postUSAValid.json

```
{
  "state":"HI"
}
```

To validate, use the *Enumeration: Post USA (Valid)* example, or using the following command.

```
validate postUSAValid.json postUSA_schema.json
```

Validation (No Matching Enumerated Value)

Guessing at the postal abbreviation for Hawaii, a reasonable guess would be *"HA"*.

Directory: chapter3, file: postUSAInvalid.json

```
{
  "state":"HA"
}
```

To validate, use the *Enumeration: Post USA (Invalid)* example, or using the following command.

```
validate postUSAInvalid.json postUSA_schema.json
```

However, the value *"HA"* is not present in the enumerated list of values for the states of the USA, so the validation process will not accept this content.

Default Values

A default value can be included in the specification of an element. When the schema specifies a default value for an element, and the element is not present in the content, then the element will be added, and the default value specified in the schema will be assigned to it.

This capability should be considered in the context of how the content will be used, to ensure proper expectations of how it will affect processing. Understanding the context is very important, as *"default"* does not have any practical use in the context of validation processing as the validation processor validates against what is present in the content only, it does not create new elements for the content.

Thus, if the only context for the schema is validation, then default values should not be used. If default values are present for other uses, it should be understood that they are ignored for validation purposes.

However, outside of validation, there are other uses of *JSON Schema* where *"default"* can be used as part of creating content.

A first use is where default is used in a message generation function, which produces *JSON* messages as part of a program or process. For example, a company issues a new policy that requires messages between divisions of a company to include a division code in each message. However, it may not be practical to retrofit all programs with additional function to handle managing division codes. Instead, the division code element is added to the general schema definition (used by validation), and specific schema definitions are created for each division (used by generation), using default to specify the division code for each division. The message generation function is then responsible for inserting the division code element with its specified default value into messages generated.

The following is a general schema for a chargeback message. It includes the division property, with a range of valid values representing the divisions of the company.

Directory: chapter3, file: chargeback_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Chargeback corporate",
  "description":"Chargeback messages for Example Inc",

  "type":"object",
  "properties":
  {
    "functionCode":{"type":"string"},
    "usageUnits":{"type":"integer"},
    "division":{"type":"integer", "minimum":10, "maximum":30}
  },
  "required":["functionCode", "usageUnits", "division"]
}
```

Using this as the starting point, schemas can be created for each division, providing the default division as part of the division specific schema. The following is such a schema, for the *Finance* division.

Directory: chapter3, file: chargebackFinance_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Chargeback finance",
  "description":"Chargeback messages for Finance division",

  "type":"object",
  "properties":
  {
    "functionCode":{"type":"string"},
    "usageUnits":{"type":"integer"},
    "division":{"type":"integer", "minimum":15, "maximum":30, "default":15}
  },
  "required":["functionCode", "usageUnits", "division"]
}
```

The constraints for the *"division"* property still contain the *"minimum"* and *"maximum"* constraints, since a program can still provide the value, and if it does then it has to be valid. The *"default"* value will fill in the *"division"* (in this case *15* for the *Finance* division) when this value is not provided by

the program. The message generated will then be valid for receipt by the destination division which can use the chargeback_schema.json general corporate schema to validate the message received.

A second use of default is in tooling. When an element has multiple selections, such as an enumeration or number range, specifying a default value can provide the tool with a default selection as part of its user interface. For example, a schema that includes an enumeration of countries could be configured with a default for the most commonly selected country.

Directory: chapter3, file: country_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Location selector",
  "description":"Central American country selection",

  "type":"object",
  "properties":
  {
    "country":
    {
      "type":"string",
      "enum":["Belize", "Costa Rica", "El Salvador", "Guatemala",
              "Honduras", "Nicaragua", "Panama"],
      "default":"Panama"
    }
  }
}
```

In this example, the user interface can display the list of Central American countries and default the selection to *"Panama"*.

JSON References (Internal) (3K)

The schema fragments used in this chapter illustrate many of the definition capabilities available through *JSON Schema*. Some of these items, like a server definition or the enumeration of states, would be useful in other schema definitions. The inclusion of references in *JSON Schema* allows definitions to be used by reference, rather than duplicating them in each schema element definition.

References can be made to definitions within the same schema definition, or to definitions in an external source. This example will use an internal reference.

The following schema definition is for a postal address in Canada. It includes the use of references for the *"province"* and *"postalCode"* elements that reference other elements within the same schema.

Directory: chapter3, file postCanadaInternal_schema.json.

```
 1 {
 2   "$schema":"http://json-schema.org/draft-04/schema#",
 3   "title":"Postal address for Canada",
 4   "description":"Postal addresses for the country of Canada",
 5
 6   "type":"object",
 7   "properties":
 8   {
 9     "address":
10     {
11       "type":"object",
12       "properties":
13       {
14         "name":{"type":"string"},
15         "number":{"type":"string"},
```

```
16          "street":{"type":"string"},
17          "street2":{"type":"string"},
18          "city":{"type":"string"},
19          "province":{"$ref":"#/definitions/CAN_province"},
20          "postalCode":{"$ref":"#/definitions/CAN_postalCode"}
21        },
22        "additionalProperties":false,
23        "required":["name", "number", "street", "city", "province", "postalCode"]
24      }
25    },
26
27    "definitions":
28    {
29      "CAN_province":
30      {
31        "type":"string",
32        "enum":["AB", "BC", "MB", "NB", "NL", "NS", "NT",
33                "NU", "ON", "PE", "QC", "SK", "YT"]
34      },
35      "CAN_postalCode":
36      {
37        "type":"string",
38        "pattern":"^[A-Z][0-9][A-Z]( )?[0-9][A-Z][0-9]$"
39      }
40    }
41  }
```

The new elements in this schema definition are,

- The addition of a *"definitions"* element at the bottom of the schema (lines 27-40). This contains the elements (*CAN_province* and *CAN_postalCode*) that can be referenced from elsewhere in the schema.
- In the definition of the properties *"province"* and *"postalCode"* (lines 19-20), instead of placing *"type":"string"* and the accompanying constraints, references are defined.

Using internal references has a number of advantages,

- It can improve the readability of the schema, especially for elements with long enumerations or complex pattern matching.
- Commonly used definitions can be separated into one place, rather then duplicated across many items. This is especially useful if the definition is subject to change, reducing the possibility of incorrectly duplicating changes across all instances.
- Ease of change if the definition is later used in a shared schema to be referenced by this and other schema definitions.

Looking at the new definition for the *"province"* property, the definition uses the keyword *"$ref"*, indicating the value associated with *"$ref"* is the URI of the schema to apply to *"province"*. The *URI* defined *"#/definitions/province"* has three parts,

- The leading # indicates the location is relative to the current schema context.
- The fragment */definitions/* is the path portion to the schema element.
- The fragment *CAN_province* is the element containing the schema content.

When the validation processing is performed, and the *"$ref"* is encountered, the validation processor will resolve the *URI* and substitute the content found at the *URI* into the location of the *"$ref"*. The content will then be validated according to the schema content from the reference.

Validation (No Errors)

To illustrate the use of references, *JSON* content for an address in the province of Prince Edward Island (postal abbreviation PE) in Canada will be used.

Directory: chapter3, file postCanadaValid.json

```
{
  "address":
  {
    "name":"John and Jane Doe",
    "number":"1124",
    "street":"Elm Street",
    "city":"Anytown",
    "province":"PE",
    "postalCode":"A1B2C3"
  }
}
```

To validate, use the *Internal Ref: Post Address (Valid)* example, or using the following command.

```
validate postCanadaValid.json postCanadaInternal_schema.json
```

Validation (Invalid Content from Referenced Schema)

To verify that the referenced schema was validated against, an invalid address is defined with a province that does not appear in the enumeration defined in the referenced schema (*"PD"* is not defined).

Directory: chapter3, file: postCanadaInvalid.json

```
{
  "address":
  {
    "name":"John and Jane Doe",
    "number":"1124",
    "street":"Elm Street",
    "city":"Anytown",
    "province":"PD",
    "postalCode":"A1B2C3"
  }
}
```

To validate, use the *Internal Ref: Post Address (Invalid)* example, or using the following command.

```
validate postCanadaInvalid.json postCanadaInternal_schema.json
```

Since the value *"PD"* is not present in the enumeration for valid *"province"* content, an error message is displayed indicating an invalid value.

JSON References (External) and the id Keyword (3L)

In the prior topic, the use of external references was mentioned. Using external references allows the common definitions to be placed in separate resources. These definitions can then be included by other schema definitions through references.

External references enable *JSON Schema* definition to reference other *JSON Schema* definitions outside the current document. The benefits of this capability include,

- Complex schemas can be composed from smaller, possibly reusable, schemas.
- Schema definitions can exist on other systems, or be accessible from the Internet.

- *URI* references are used, allowing access to be made using different protocols, storage systems, or messaging systems.

However, there are some operational aspects that need to be considered.

- Remote access to schemas may be compromised by network failures, provider failures, or degraded network performance. Use of caches, alternate hosting servers, or similar mechanisms may be used to mitigate these issues.
- Version control may need to be incorporated into the definition, storage, and access for the schema content.
- *URI* resolution and fetch logic may need to be incorporated into the validation processors that will process schemas that include external references. Two approaches will be shown in this topic.

Use of the Id Keyword

To make content addressable to external access, *URI* definitions are included in the schema definitions using the *"id"* keyword. This *URI* information is required to enable the schema resolver to locate the content being referenced.

The use of *"id"* element is flexible, it can be used at the root of the schema and/or within elements of the schema. Where to use *"id"* is dependent on how the schema is intended to be used. For this example, the use of an external schema to hold common definitions, a top level *"id"* definition is suitable.

As a practical note, schema processors need to have a base *URI* for every schema that will be referenced. Therefore, it makes sense to always have a top level *"id"* element defined. Anywhere a *URI* is expected in a validation processor function/method, this top level *"id"* will be used (it is fairly common for schema processors to look for this top level *"id"* when the *URI* is not provided).

Directory: chapter3, file postCanadaCommon_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Canada provincial postal abbreviations",
  "description":"Postal abbreviations for Canada",

  "id":"http://localhost:8081/schema/postCanadaCommon_schema.json",

  "definitions":
  {
    "CAN_province":
    {
      "type":"string",
      "enum":["AB", "BC", "MB", "NB", "NL", "NS", "NT",
              "NU", "ON", "PE", "QC", "SK", "YT"]
    },
    "CAN_postalCode":
    {
      "type":"string",
      "pattern":"^[A-Z][0-9][A-Z]( )?[0-9][A-Z][0-9]$"
    }
  }
}
```

The schema definition is similar to any other, except that the content is fragments of schema content rather than a complete schema definition. The other addition is the *"id"* element in the sixth line, declaring the *URI* for the schema. The elements declared within the schema are referenced relative to

this *URI*, so for example, *http://localhost:8081/schema/postCanadaCommon_schema.json/definitions/CAN_province* will be used in the *"$ref"* element of a schema definition referencing the *CAN_province* element.

Schema Referencing an External Schema

Using an equivalent to the local reference example, the following is a Canadian postal address schema definition that references the external schema definitions.

Directory: chapter3, file: postCanada_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Postal address for Canada",
  "description":"Postal address for Canada",

  "type":"object",
  "properties":
  {
    "address":
    {
      "type":"object",
      "properties":
      {
        "name":{"type":"string"},
        "number":{"type":"string"},
        "street":{"type":"string"},
        "street2":{"type":"string"},
        "city":{"type":"string"},
        "province":
        {
            "$ref":"http://localhost:8081/schema/postCanadaCommon_schema.json#/
definitions/CAN_province"
        },
        "postalCode":
        {
            "$ref":"http://localhost:8081/schema/postCanadaCommon_schema.json#/
definitions/CAN_postalCode"
        }
      },
      "additionalProperties":false,
      "required":["name", "number", "street", "city", "province", "postalCode"]
    }
  }
}
```

The use of *"$ref"* for the *"province"* and *"postalCode"* elements specify a *URI* that is not within the current schema, since there is content preceding the #. This portion of the *URI* (in this case *http://localhost:8081/schema/postCanadaCommon_schema.json*) is used by the schema processor to locate the referenced content.

The behavior of the schema processing is the same as for a local reference, only the retrieval of the schema content differs. How the schema content from the external schema arrives has many choices. Three different options are shown in the remainder of this topic.

The Role of the Schema Processor

URI resolution is performed by the schema processor. *URI* definitions can be associated with any type of resource including web resources, file systems, content within a data store, or content that is

dynamically generated. The schema processor is responsible for determining the location the *URI* corresponds to and the manner in which interaction with that location is conducted.

The schema definition itself does not need to know how the schema processor works, or if the interactions between the schema processor and the resource are the same or different each time. For instance, cached content can be used when the same resource is accessed multiple times.

While public web addresses can be used for the *URIs*, for many deployment cases this will not be suitable. Local instances of the schema content accessible through reliable networks, with version control if needed, are more suitable for production use. In addition, *URI* definitions are not limited to web servers and may be used to support use of storage facilities such as databases, or flexibility to use different persistent storage options by deployment configuration.

To illustrate these different deployment approaches, three deployment models will be shown.

- The first will use a local web server as the resource manager of the external schema.
- The second will use the local file system as the resource manager.
- The third will use a database.

From these examples, other types of resource managers can be envisioned.

Web Based Schema Resource Manager

A local web server will be used to provide access to the external schema over the *HTTP* protocol. Both *Node.js* and *Python* provide a simple web server capability in their default runtime libraries, and either of these can be used for this example.

In the example, the *URI* includes *"http://localhost:8081"* which will be used for the configuration of the local web server. If an external web server is used, this part of the *URI* would be updated with the appropriate address content (e.g., *"http://www.example.com"*). The port number *8081* is unlikely to be already in use, but if it is, any unused port number can be used.

To start the web server. Open a *Terminal / Command Prompt* and change directory to the install location for the book examples (*~/bookujs* and *c:\bookujs* are used in the commands that follow, update these if necessary to match your install location).

For *Node.js* on *Linux*, use the following commands,

```
cd ~/bookujs/chapter3/web
node tinyserver.js 8081
```

For *Node.js* on *Windows*, use the following commands,

```
cd c:\bookujs\chapter3\web
node tinyserver.js 8081
```

For *Python 3.x* on *Linux*, use the following commands,

```
cd ~/bookujs/chapter3/web
python -m http.server 8081
```

For *Python 3.x* on *Windows*, use the following commands,

```
cd c:\bookujs\chapter3\web
python -m http.server 8081
```

For *Python 2.x* on *Linux*, use the following commands,

```
cd ~/bookujs/chapter3/web
python -m SimpleHTTPServer 8081
```

For *Python 2.x* on *Windows*, use the following commands,

```
cd c:\bookujs\chapter3\web
python -m SimpleHTTPServer 8081
```

This command will start the web server on the port indicated (*8081*), with the web directory under *chapter3* as the base directory for the web server content. In this directory is the schema directory and a copy of the file containing the externally referenced schema content (*postCanadaCommon_schema.json*).

When finished running the programs that access the web server content, the web server can be terminated. Switch to the *Terminal / Command Prompt* window running the web server, and press *Ctrl-C* to terminate it. The *Terminal / Command Prompt* window can then be closed.

To run the validation program including retrieval of this content from the web server, open another *Terminal / Command Prompt* and in the *chapter3* directory use the following command.

```
validate postCanadaValid.json postCanada_schema.json
```

The valid message will be displayed.

To verify that the external schema is being accessed and validated against, run the validation against the invalid address using the following command.

```
validate postCanadaInvalid.json postCanada_schema.json
```

This will display the message that the province specified in the address is not valid. Since the enumeration is only present in the external schema, this verifies that the external schema was used for the validation.

File System Based Schema Resource Manager

In the web server example, the external schema content was retrieved by an *HTTP* request from a web server. The external schema content itself was in a file accessible to the web server. In this example, the schema content will be loaded from the file system directly; using the same *URI* as in the web example. This will be accomplished by pre-loading the external schema content.

This example shows how network dependencies can be eliminated when external schemas are used. The schemas can be copied to a local file system, and the schema processor can be populated with the schemas from the local source. All the *URIs* stay the same, so none of the schemas need to be modified. This ensures schema versions can be verified using comparison, and avoids inadvertent typographical errors.

To make the external schema content known to the schema processor, the list of schema files will be included in the command line arguments. The schemas will then be loaded from the local file system and populated into the schema cache.

Note: the *validate* program (both *Javascript* and *Python* versions) is covered in *chapter 8*, where the implementation of the schema management function is detailed.

In many of the examples, the schema processor has been shown using a command similar to the following,

```
validate simpleArray.json simpleArray_schema.json
```

For this implementation of external references, the additional resources are specified in the command to start the *validate* program. The first two arguments are the same (*JSON* content file and *JSON Schema* file). Following these, is the list of files containing referenced schema content. For example,

```
validate example.json example_schema.json common_schema.json
```

Path information can also be used in the command. For example, if the externally referenced content is in a shared location (e.g., a directory named common), the command for *Linux* would be,

```
validate example.json example_schema.json ../common/common_schema.json
```

For *Windows*, it would be,

```
validate example.json example_schema.json ..\common\common_schema.json
```

When the schema processor receives arguments that include external reference sources, it will perform the following actions.

- The *JSON* content file and *JSON Schema* file will be read and each validated for correct *JSON syntax*.
- For each reference schema, the file will be read, validated for correct *JSON syntax*, and added to the schema content for the schema validation.
- The schema validation will be performed (*Tiny Validator* or *jsonschema*).
- The valid response or error response and information will be handled.

The *URI* resolution for the $ref and id elements are resolved within the schema content, the naming and organization of the file resources does not impact the resolution of the schema references. This allows the file resources to be placed freely, without impacting the schema content.

To run this example, the same *JSON* content and *JSON Schema* files will be used as the prior example. However, this time the web server that served the external schema content will no longer be running (if the web server is still running from the prior example, shut it down before running this example to show that it is not used). By using this as the example, the use case described in the opening of this example is demonstrated.

To run the validation of a valid address, use the following command,

```
validate postCanadaValid.json postCanada_schema.json postCanadaCommon_schema.json
```

The valid message will be displayed.

To verify that the external schema is being accessed and validated against, run the validation against the invalid address using the following command.

```
validate postCanadaInvalid.json postCanada_schema.json postCanadaCommon_schema.json
```

The message will display that the value for province is not valid, the same error as the preceding web example.

With the same commands as used for the web server example, the example has shown that the same schemas can be used with different schema resource management approaches to achieve the same processing results.

JSON Schema Database Resource Manager

Managing schemas is not limited to files, they can also be stored using other approaches. A simple database, using *JSON* of course, is used to show this. The name *jsbd* (*JSON Schema Database*) will be used.

- The persistent storage will be in the file *jsdb.json*. This file is located in the *tools/node/validate* and *tools/python/validate* directory (same file content in each).
- The *URI* for an external reference is *"jsdb:aaa#bbb"* where *jsdb*: is the resource, *aaa* is the location and *bbb* is the fragment within the schema.

- Nothing special is required in the *validate* command syntax, this feature will be used for any references using the *"jsdb:" URI*.

To show the use of the *jsdb* database, the postal address example will be modified to use the database to store the schema content.

Note: the source code for the *validate* program (both *Javascript* and *Python* versions) is shown in *chapter 8*, where the implementation of the pre-loading function is detailed.

The schema file is updated to use the new *URI*, but the *JSON* content file remains the same.

Directory: chapter3, file: postCanada_jsdb_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Postal address for Canada",
  "description":"Postal address for Canada",

  "type":"object",
  "properties":
  {
    "address":
    {
      "type":"object",
      "properties":
      {
        "name":{"type":"string"},
        "number":{"type":"string"},
        "street":{"type":"string"},
        "street2":{"type":"string"},
        "city":{"type":"string"},
        "province":{"$ref":"jsdb:postCAN#/definitions/CAN_province"},
        "postalCode":{"$ref":"jsdb:postCAN#/definitions/CAN_postalCode"}
      },
      "additionalProperties":false,
      "required":["name", "number", "street", "city", "province", "postalCode"]
    }
  }
}
```

The content is the same except for the $ref values for *province* and *postalCode*. The *URI* content now starts with *jsdb*: to indicate the resource manager and the location is specified as *postCAN* since a database *URI* is a key style resource rather than a file style resource. The remainder of the *URI* (starting with #) is the same, referencing the same content within the schema.

To run the validation of a valid address, use the following command,

```
validate postCanadaValid.json postCanada_jsdb_schema.json
```

The valid message will be displayed.

To verify that the external schema is being accessed and validated against, run the validation against the invalid address using the following command.

```
validate postCanadaInvalid.json postCanada_jsdb_schema.json
```

As with the prior two examples, the value for *province* is displayed as not valid for the schema.

As will be discussed in greater detail in *chapter 8*, the database schema manager only loads the schemas that are needed for the schema being processed. Thus the *jsdb* database could contain a large number of schemas, but the schema processor will only fetch and load the schemas actually used. This can ease management of large deployments, while ensuring efficient processing.

Special note: This example used a new *URI* structure (*jsdb*:) to illustrate the flexibility offered by the use of *URIs* for references. If preserving the original *URI* was required, this could be achieved by using a *URI* mapper to identify those *URIs* to fetch from the database instead of from the original location. This is similar to the pre-load approach in the prior example, but using the database as the pre-load source rather than the file system.

Nested Reference Schema Definitions

Schema definitions can contain references, including schema definitions that are references. Thus, schema definitions can have many layers. This extends the concept of schema reuse, enabling schema definitions to be assembled in many forms. An example is provided in the person example, where a three layer schema definition is used. The example uses a *JSDB* resource, which makes the layered definition easy to see.

Directory: chapter3, file: person_jsdb.json

```
[
  {
    "$schema":"http://json-schema.org/draft-04/schema#",
    "title":"Person",
    "description":"Simple person definition",

    "id":"jsdb:person",

    "type":"object",
    "properties":
    {
      "name":{"type":"string"},
      "address":{"$ref":"jsdb:address#"}
    },
    "required":["name", "address"]
  },
  {
    "$schema":"http://json-schema.org/draft-04/schema#",
    "title":"Address",
    "description":"Simple address",

    "id":"jsdb:address",

    "type":"object",
    "properties":
    {
      "street":{"type":"string"},
      "city":{"type":"string"},
      "zipcode":{"$ref":"jsdb:zipcode#/definitions/zipcode"}
    },
    "required":["street", "city", "zipcode"]
  },
  {
    "$schema":"http://json-schema.org/draft-04/schema#",
    "title":"Zip code",
    "description":"Zip code",

    "id":"jsdb:zipcode",

    "definitions":
    {
      "zipcode":
      {
        "type":"string",
```

```
          "pattern":"^[0-9]{5}(-[0-9]{4})?$"
        }
      }
    }
  ]
```

The first entry (with id *jsdb:person*) includes a property, *address*, that is defined through a reference (*$ref jsdb:address#*).

The second entry (with id *jsdb:address*) includes a property, *zipcode*, that is defined through a reference (*$ref jsdb:zipcode#/definitions/zipcode*).

The third entry (with the id *jsdb:zipcode*) contains the definition to be applied to the *zipcode* property.

When the schema processor encounters a reference for *jsdb:person*, it will load the corresponding schema (the first entry). This will surface a new dependency, the *jsdb:address* reference, which when loaded will surface the last dependency, *jsdb:zipcode*. This process is recursive, with each layer resolved, the schema processor needs to determine if any new dependencies were introduced, and if so, resolve those. The following is an example of a simple schema that contains a reference to the *jsdb:person* schema.

Directory: chapter3, file: person_jsdb_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Person",
  "description":"Person, for multi level references example",

  "type":"object",
  "properties":
  {
    "person":{"$ref":"jsdb:person#"}
  },
  "additionalProperties":false,
  "required":["person"]
}
```

To show validation, a valid *JSON* document is shown first.

Directory: chapter3, file: personValid.json

```
{
  "person":
  {
    "name":"John Doe",
    "address":
    {
        "street":"123 Oak St",
        "city":"Anytown",
        "zipcode":"12345"
    }
  }
}
```

To run the validation, use the following command.

```
validate -j=person_jsdb.json personValid.json person_schema.json
```

To show that all the layers were resolved, an invalid *JSON* document contains a zip code that does not conform to the constraint defined in the lowest layer schema.

Directory: chapter3, file: personInvalid.json

```
{
  "person":
  {
```

```
      "name":"John Doe",
      "address":
      {
          "street":"123 Oak St",
          "city":"Anytown",
          "zipcode":"12345ABC"
      }
    }
  }
```

To run the validation, use the following command.

```
validate -j=person_jsdb.json personValid.json person_schema.json
```

This example is simplified to focus on the depth mechanism. However, the benefits of granularity control and flexible schema composition allow the value of this mechanism to be easily imagined for complex schema libraries.

Other Resource Managers

General purpose databases, source code control systems, or specialized schema management systems are all possible locations where schema content can be stored and accessed from. Each alternative provides its own features and benefits suited to different needs.

The resource manager can vary in implementation to suit the particular system needs. As shown in this chapters examples, the level of abstraction, order of loading, and storage / access method are all design decisions for the implementer of the resource manager.

The standards activities for *JSON Schema* do not include definition of interfaces for validation libraries or programming languages, so using custom resource managers requires programs to program to the interfaces of the resource managers and validation library chosen. *Chapter 8* shows two different implementations using different resource managers, each suited to the libraries and programming languages used.

Using External References in the JSON Validate Web Tool

The *JSON Validate* web tool includes a *References* section which allows up to 8 schema definitions to be included with the primary schema.

The examples provided use the same content as the earlier web and file resource examples.

- The *External Ref: Post Address (Valid)* example will populate the schema, content and first reference fields in the tool. Press the *Validate* button to show the valid result.
- To show an error result, load the *External Ref: Post Address (Invalid)* example and press the *Validate* button.

To use more than one reference schema in the *JSON Validate* tool, select from the numbered tabs in the *References* section, and enter content into each tab needed. If any of the reference schema content is invalid, the tabs that contain errors will be highlighted with a red border.

4. Conditional Content

What if the portions of the *JSON* content are dependent on the value of one of its elements? There are many examples, and different approaches that can be considered. Some examples,

- An object has four properties, of which two are always required and two are optional. The *required* keyword provides an efficient expression of this constraint.
- An object has four properties, but two of those properties are mutually exclusive. A combination of the *oneOf* and *required* keywords can express this.
- An object has properties that are only present if a particular property is present. The *dependencies* keyword can be used to define this constraint.
- A group of properties is dependent on the value of a selector property. Using an object definition with an *enumeration* and *oneOf* can be used to define this relationship.

These example can be combined to express a wide variety of conditional content constraints. A key enabler of this flexibility is the many places where schema definitions can be embedded within the schema itself. The *additionalProperties* examples in *chapter 3* showed both the boolean option and the schema option for the constraint.

Mutually Exclusive Properties

Designing a schema where properties are mutually exclusive can utilize the *oneOf* constraint to prevent conflicting properties being present in the same content.

In this example, the a schema for taxpayer identification in the United States of America is defined. A particular entity will only have one identifier, although that could be one of several types.The property name is always required, while only one of the properties *ssn* (*Social Security Number*), *ein* (*Employer Identification Number*) or *itin* (*Individual Taxpayer Identification Number*) is allowed.

Directory: chapter4, file taxEntity_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Taxation Id",
  "description":"Identification number for taxation purposes",

  "type":"object",
  "properties":
  {
    "name":{ "type":"string" },
    "ssn":{ "type":"string", "pattern":"^[0-9]{3}-[0-9]{2}-[0-9]{4}$" },
    "ein":{ "type":"string", "pattern":"^[0-9]{2}-[0-9]{7}$" },
    "itin":{ "type":"string", "pattern":"^9[0-9]{2}-[0-9]{2}-[0-9]{4}$" }
  },
  "required":["name"],
  "additionalProperties":false,

  "oneOf":
  [
    { "required":["ssn"] },
    { "required":["ein"] },
    { "required":["itin"] }
  ]
}
```

The object definition contains all the possible properties. The *name* property is defined in the *required* constraint with the object definition. A separate *oneOf* constraint is defined at the same level as the object definition. It uses the *oneOf* constraint with an enclosed array of *required* constraints, indicating that one, and only one, *required* constraint must be met. This ensures that one of the identifiers is present, and that no combination of two or more is present.

Since the identifier type is not conditional on another property, and the object is not too busy when listing the presented options, using this definition approach is simple to comprehend and use. If there were a large number of options or other properties, creating a hierarchical holding object would be a design consideration.

To test the example, valid (*taxEntityValid.json*) and invalid (*taxEntityInvalid.json*) examples are provided in the *chapter4* directory. These can be used with the *validate* command line tool using the *taxEntity_schema.json* schema. In the *JSON Validate* web tool, import the examples *4A Mutually Exclusive: Tax Entity (Valid)* and *4A Mutually Exclusive: Tax Entity (Invalid)*.

Dependent Properties

In the *order* example (*chapter 3*, *Dependencies for Properties (3G)*), the *dependencies* constraint was used to express the relationship between the properties for the shipping information and the properties for the loyalty program. The example enabled the schema to require the presence of a property when another property was present.

The *dependencies* constraint can also be defined using a schema, allowing conditional content definition through the constraint.

In the following modified version of the *order* example, the dependency for the *shipTo* property is defined as a schema containing two properties (*shipAddress* and *signature*). The *order* schema content does not include these two properties, instead they are declared within the dependency schema definition. The properties are only introduced when the *shipTo* property is used.

Directory: chapter4, file order2_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Order",
  "description":"Order billing and shipping information",

  "type":"object",
  "properties":
  {
    "orders":
    {
      "type":"array",
      "items":
      {
        "properties":
        {
          "order":{"type":"string"},
          "billTo":{"type":"string"},
          "billAddress":{"type":"string"},
          "shipTo":{"type":"string"}
        },
        "required":["order", "billTo", "billAddress"],
        "dependencies":
        {
          "shipTo":
```

```
            {
              "properties":
              {
                "shipAddress":{"type":"string"},
                "signature":{"type":"boolean"}
              },
              "required":["shipAddress","signature"]
            }
          }
        }
      }
    }
  }
```

To test the example, valid (*orderValid2.json*) and invalid (*orderInvalid2.json*) examples are provide in the *chapter4* directory. These can be used with the *validate* command line tool using the *order2_schema.json* schema. In the *JSON Validate* web tool, import the examples *4B Dependent Properties: Order (Valid)* and *4B Dependent Properties: Order (Invalid)*.

Selector Driven Schemas

When conditional content is associated with the value of a property (a selector), a selectable content pattern can be applied.

For instance, portions of international mailing addresses are dependent on the country the address is in. These include the names of the elements (e.g., province, state, postal code, zip code) and the format of the elements (e.g., varying length and structure of postal codes).

The following example shows a schema covering addresses for the North American countries of Canada, United States of America, and Mexico.

To improve readability, the schema is shown in two sections. The second section contains the province/state and postal code/zip code definitions.

Directory: chapter4, file postNorthAmerica_schema.json

Breaking the first part of the schema down into sections,

- Lines 1-8 are the standard leading content for *JSON* content with an object as the root element.
- Lines 9-13 define the *"address"* element as an object.
- Lines 14-18 define the properties for the address that are present in all forms of a North American address.
- Line 19 introduces the divergence of properties, using the name *"national"* to indicate the context of the divergence.
- Line 21 uses the *JSON Schema* keyword *"oneOf"* which indicates that one, and only one, of the following must be satisfied by the *JSON* content being validated. *"oneOf"* is an array type, and what follows is an array containing the elements that will be compared.
- Lines 23-32 contain the definition for a valid Canada address.
- Lines 33-42 contain the definition for a valid United States of America address.
- Lines 43-52 contain the definition for a valid Mexico address.
- Lines 53-59 finish the definition with the *additionalProperties* and *required* constraints.

Local references are used to the definitions of state/province and postal code/zip code for each country. These definitions are on lines 61-104, covered in the second section.

The element that acts as the selector in the schema is the *"country"* element (lines 26, 36 and 46). Note that each country element is an enumeration with a single value. This element must be present in every address, and acts as the selector for the country specific content present in each *"national"* element.

```
 1 {
 2   "$schema":"http://json-schema.org/draft-04/schema#",
 3   "title":"North America address",
 4   "description":"Postal addresses for Canada, USA and Mexico",
 5
 6   "type":"object",
 7   "properties":
 8   {
 9     "address":
10     {
11       "type":"object",
12       "properties":
13       {
14         "name":{"type":"string"},
15         "number":{"type":"string"},
16         "street":{"type":"string"},
17         "street2":{"type":"string"},
18         "city":{"type":"string"},
19         "national":
20         {
21           "oneOf":
22           [
23             {
24               "properties":
25               {
26                 "country":{"type":"string", "enum":["CAN"]},
27                 "province":{"$ref":"#/definitions/CAN_province"},
28                 "postalCode":{"$ref":"#/definitions/CAN_postalCode"}
29               },
30               "additionalProperties":false,
31               "required":["country", "province", "postalCode"]
32             },
33             {
34               "properties":
35               {
36                 "country":{"type":"string", "enum":["USA"]},
37                 "state":{"$ref":"#/definitions/USA_state"},
38                 "zipCode":{"$ref":"#/definitions/USA_zipCode"}
39               },
40               "additionalProperties":false,
41               "required":["country", "state", "zipCode"]
42             },
43             {
44               "properties":
45               {
46                 "country":{"type":"string", "enum":["MEX"]},
47                 "state":{"$ref":"#/definitions/MEX_state"},
48                 "postalCode":{"$ref":"#/definitions/MEX_postalCode"}
49               },
50               "additionalProperties":false,
51               "required":["country", "state", "postalCode"]
52             }
53           ]
54         }
```

```
      },
      "additionalProperties":false,
      "required":["name", "number", "street", "city", "national"]
    }
  },
```

The second part of the schema specifies the province / state list for each country and the postal code / zip code patterns for each. The definition names start with the country prefix, keeping the schema names unique even when different countries use the same name for the individual elements.

```
  "definitions":
  {
    "CAN_province":
    {
      "type":"string",
      "enum":["AB", "BC", "MB", "NB", "NL", "NS", "NT",
              "NU", "ON", "PE", "QC", "SK", "YT"]
    },
    "CAN_postalCode":
    {
      "type":"string",
      "pattern":"^[A-Z][0-9][A-Z]( )?[0-9][A-Z][0-9]$"
    },
    "USA_state":
    {
      "type":"string",
      "enum":["AL", "AK", "AR", "AS", "AZ", "CA", "CO", "CT",
              "DC", "DE", "FL", "FM", "GA", "GU", "HI", "IA",
              "ID", "IL", "IN", "KS", "KY", "LA", "MA", "MD",
              "ME", "MH", "MI", "MN", "MO", "MP", "MS", "MT",
              "NC", "ND", "NE", "NH", "NJ", "NM", "NV", "OH",
              "OK", "OR", "PA", "PR", "PW", "RI", "SC", "SD",
              "TN", "TX", "UT", "VA", "VI", "VT", "WA", "WI",
              "WV", "WY", "AA", "AE", "AP"]
    },
    "USA_zipCode":
    {
      "type":"string",
      "pattern":"^[0-9]{5}(-[0-9]{4})?$"
    },
    "MEX_state":
    {
      "type":"string",
      "enum":["AGS", "BC", "BCS", "CAM", "COAH", "COL", "CHIH",
              "CHIS", "DF", "DGO", "GTO", "GRO", "HGO", "JAL",
              "MEX", "MICH", "MOR", "NAY", "NL", "OAX", "PUE",
              "QRO", "Q ROO", "SLP", "SIN", "SON", "TAB",
              "TAMPS", "TLAX", "VER", "YUC", "ZAC"]
    },
    "MEX_postalCode":
    {
      "type":"string",
      "pattern":"^[0-9]{5}$"
    }
```

When a validation processor is checking the *JSON* content, resolution of the *"country"* element provides the validation process with the definition to apply for the remainder of that element. This includes the names of the elements (e.g., *"province"* or *"state"*) and constraints to apply to the element. For instance, Canada and Mexico both use the name *"postalCode"*, but in Canada this is a mixed alphanumeric that is 6 or 7 characters long, whereas in Mexico it is a 5 digit number.

To test the example, valid (*postCanadaValid.json*) and invalid (*postCanadaInvalid.json*) examples are provide in the *chapter4* directory. These can be used with the *validate* command line tool using the *postNorthAmerica_schema.json* schema. In the *JSON Validate* web tool, import the examples *4C Selector Driven Content: Address (Valid)* and *4C Selector Driven Content: Address (Invalid)*.

Alternative Implementations for Selector Driven Schemas

There are other ways to address conditional content, that may be applicable in different use cases. To illustrate some of these, a database configuration example will be used.

Many programs use databases, and a desirable feature for many of these programs is user selection of which database to use. However, each database has its own configuration options such as naming and parameters for connectivity.

If the approach shown above for the North American address was used, instead of using country as the selector, the database name and version would be the selector. The configuration data for each database choice would be encapsulated within this construct.

Other options for addressing this include isolated schemas, generic database configuration definitions, or a comprehensive list of elements.

Isolated Schema Definitions

This is a direct derivative to the selector driven example, using separate schema definitions for each selection.

Instead of placing the database definition in the main schema, the main schema could contain the key information (in this case, database and version), with a reference to separate schema definitions for each database. Each separate schema would contain database specific elements.

This has the advantage of being extensible by populating the schema repository with additional database schema definitions without updating the base schema. The disadvantage is the additional schema management.

Separate Schema Definitions

Copies of the full schema can be created that each include the specific content for each database. The name of the schema would then act as the selector. For instance,

- *program_db_a_3_schema.json* for version 3 of database product A.
- *program_db_b_5_schema.json* for version 5 of database product B.

The advantage is a simple schema for each database configuration. The disadvantages include,

- Changes to common elements require cascading the changes through every definition.
- Programs needs to keep a registry of schema names or have a static naming convention.
- Additional schema management required.

Generic Database Configuration

In this approach, the conditional constraint is not used. Instead, a generic database definition is created that can be applied to all possible databases. The program using the configuration data will then be responsible for translating the generic data into a form acceptable for the database being used.

The advantage to this approach is that the schema is the same regardless of database to be used. The two main disadvantages are,

- The person writing programs that use the schema needs to understand how to translate the generic definition into the specific database data.
- The use of database specific features may be limited by the lack of data provided.

Comprehensive Configuration

Instead of abstracting away as the generic definition does, the comprehensive definition provides a broad list of elements. Each database fills in those elements that are relevant to itself. Like the generic approach, this has the advantage of having a single base definition. However, like the generic approach, the program using the definition needs to know what elements to use and which to ignore. This approach also loses the ability for validation to find many configuration errors since the relationships between elements don't have the context of database specific schema definitions.

Uses for the Conditional Content Approaches

The techniques for defining schema definitions to address conditional content can be used in a variety of ways. The selection of technique will be driven by a combination of functionality, readability, and usability. There is often a choice in approach, and the weight of factors to consider for different schemas may vary. A schema intended for broad infrequent use may weight readability heavily, whereas an optimized business to business transaction schema may accept more complexity for more precise constraint definition.

As shown in the address example, complex data types that vary in implementation can be factored into common and unique elements. Validation constraints can then be associated with each instance of the unique elements. This approach can also be used for managing content differences between versions of a schema. Common elements can be shared, whereas version specific elements can be isolated using a version identifier to encapsulate version specific content.

5. Configuration Files

Configuration files are often a good candidate for using *JSON* and *JSON Schema*.

- Typically, configuration files are read and written as a whole, which fits well with file based storage and databases that store *JSON* documents.
- The content storage format and the in-memory representation of the content are well aligned, whether the programming language is Javascript or not. Serialization and deserialization mechanisms are available, allowing the persistent form to be used across programs using different languages and/or runtime platforms.
- Configurations are often intended to be human readable, and editable, outside of the execution of the program using the configuration. Since *JSON* is a text format, it supports this activity.
- Validation of content is practical, and can be performed both by the program consuming the content and by independent validation processes. This augments the benefit of human editing activity, since the content can be validated as part of the editing process.

Configurations may be single files or multiple files, the design of the individual system can determine the appropriate implementation approach. The use of files in a file system is shown in the example, however other equivalent persistent storage options can be used (e.g., database support *JSON* content, database with arbitrary content field support (e.g., BLOB), cloud storage repository, or similar).

Example Configuration File

A system that has multiple server programs on a single physical or virtual server instance needs to ensure that each server program is given its own IP port to prevent conflicts. The IP port assignments also must not conflict with any existing ports being used, or use any of those planned for future use. The use of a configuration file, rather than limiting port choice to hard coding, to enable assignment of the IP ports is therefore useful.

The files for this example are found in the *chapter5* directory. This directory contains the configuration file, the configuration schema, and three each of *Javascript* and *Python* program files.

To begin, the server definitions for two servers are shown in the following configuration file, *startup.json*.

Directory: chapter5, file: startup.json

```
{
  "servers":
  [
    {
      "name":"Web Server",
      "start":true,
      "program":"webserver",
      "port":8301
    },
    {
      "name":"Data Server",
      "start":true,
      "program":"dataserver",
      "port":8302
```

```
      }
    ]
  }
```

The schema for the startup configuration content supports defining an array of server definitions.

Directory: chapter5, file: startup_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Server start up",
  "description":"List of servers to start",

  "type":"object",
  "properties":
  {
    "servers":
    {
      "type":"array",
      "items":
      {
        "type":"object",
        "properties":
        {
          "name":{"type":"string"},
          "start":{"type":"boolean"},
          "program":{"type":"string"},
          "port":{"type":"integer"}
        },
        "additionalProperties":false,
        "required":["name", "start", "program", "port"]
      }
    }
  }
}
```

While the content shows two servers, the schema defines an array without a limit of two elements, allowing additional servers to be defined if desired.

Programs Consuming the Configuration File

The program that consumes the configuration file is shown next (*Javascript/Node.js*, then *Python* versions). It reads the configuration file, and processes each of the servers definition present.

Directory: chapter5, file: startup.js

```
 1 /*
 2  * Server launcher
 3  */
 4 var fs = require ("fs");
 5 var fork = require ("child_process").fork;
 6
 7 //if module invoked directly, call main
 8 if (require.main === module) {
 9     main ();
10 }
11
12 /**
13  * Load configuration and initiate server launches.
14  */
15 function main () {
16     console.log ("Reading configuration from startup.json.");
17     // load configuration file, with server startup data
18     var configuration = null;
```

```
19      try {
20          var data = fs.readFileSync ("startup.json");
21          configuration = JSON.parse (data);
22      } catch (e) {
23          console.log ("Error loading configuration: " + e.message);
24          process.exit (1);
25      }
26      // start servers
27      launchServers (configuration.servers);
28  }
29
30  /**
31   * Launch all servers marked with start:true.
32   * @param {object} servers List of servers to launch
33   */
34  function launchServers (servers) {
35      // for each server in configuration
36      for (var ctr = 0; ctr < servers.length; ctr++) {
37          var server = servers[ctr];
38          // if server marked to start
39          if (server.start) {
40              // populate port number in args, start child process
41              console.log ("Starting " + server.name);
42              var args = ["--port=" + server.port];
43              fork (server.program + ".js", args, null, startError);
44          }
45      }
46  }
47
48  /**
49   * On error starting a server, display message, terminate program.
50   * @param {string} error Error message
51   */
52  function startError (error) {
53      console.log ("Error starting server: " + error);
54      process.exit (1);
55  }
```

In the *Javascript/Node.js* version of the startup program,

- In the *main* function, lines 20-21 load the configuration data from the *startup.json* file, parse it. If either of these throw an exception, lines 23-24 print a message and terminate the program. Otherwise, the function to start the servers is called (line 27).
- In the *launchServers* function, line 36 walks through the list of servers defined in the configuration file. Line 39 determines if the server is to be started (start is *true*), and if so, lines 41 to 43 will display a message then start a child process as specified by the server definition. The fork library call accepts a function to call if an error occurs starting the process, which is the *startError* parameter in this example.
- The *startError* function, lines 52-55, prints a message then terminates the program.

Since the processes created with this program are child processes of the startup program and the child process output is not otherwise captured, the output of the child programs will be displayed along with the parent program. When the parent program is terminated (pressing *Ctrl-C*), the child processes will be terminated as well.

Directory: chapter5, file: startup.py

```
1  """
2  Server launcher
3  """
```

```
 4 from json import loads
 5 from subprocess import Popen
 6 import sys
 7 from time import sleep
 8
 9 def main ():
10     print ("Reading configuration from startup.json.")
11
12     # load configuration file, which contains the startup
13     # directions for all servers.
14     try:
15         # read the file and convert to a JSON object
16         data = open ("startup.json", "rU").read ()
17     except IOError as e:
18         print ("Error reading configuration file: " + e.strerror)
19         sys.exit (1)
20
21     try:
22         configuration = loads (data)
23     except Exception as e:
24         print ("Invalid JSON content in startup.json")
25         sys.exit (1)
26
27     launchServers (configuration["servers"])
28
29 def launchServers (servers):
30     # for each server in configuration
31     processes = []
32     for server in servers:
33         # if server marked to start
34         if server["start"]:
35             # set port number in arguments and start child process
36             print ("Starting " + server["name"])
37             program = server["program"] + ".py"
38             port = "--port=" + str (server["port"])
39             process = Popen (["python", program, port])
40             processes.append (process)
41             # allow child process messages to display
42             sleep (0.25)
43
44     for process in processes:
45         process.wait ()
46
47 if __name__ == "__main__":
48     main ()
```

For the *Python* version of the startup program, in the *main* function,

- Lines 14-19 load the configuration data from the *startup.json* file, printing a message and terminating the program if an error occurs.
- Lines 21-25 parse the loaded data, printing a message and terminating the program if an error occurs.
- Line 27 calls the *launchServers* function with the list of servers to start.

In the *launchServers* function, a mechanism is used to keep the main process active while the child processes are running. This will then allow termination of the main process to also terminate the child processes.

- Line 31 creates a list to hold the set of processes created, line 40 adds each process created to the list, and lines 44-45 have the parent process wait on the child processes to all end. The termination of the parent process will also then terminate the child processes.
- Line 32 walks through the list of servers defined in the configuration file. Line 34 determines if the server is to be started (start is *true*), and if so, lines 36 to 40 will display a message then start a child process as specified by the server definition.
- Line 42 adds a short (¼ second) delay between starting child processes to prevent overlapping display output from the child processes. (this is just a convenience for the example display output, it is not necessary for other implementations).

Since the processes created with this program are child processes of the startup program and the child process output is not otherwise captured, the output of the child programs will be displayed along with the parent program. When the parent program is terminated (pressing *Ctrl*-C), the child processes will be terminated as well.

Executing the Startup Programs

The programs started by the startup program in this example (*webserver* and *dataserver*) are *Javascript/Node.js* and *Python* programs corresponding with the runtime of the example. However, this is only a convenience for the example, child processes using other technologies can also be started using the same library functions from each runtime.

To start the *Javascript/Node.js* version of the program, the following command can be used in the *chapter5* directory in a *Terminal* or *Command Prompt* window.

```
node startup.js
```

To start the *Python* version of the program use

```
python startup.py
```

The program will start two child processes, one containing the web server and one containing the data server. While the program logic for the two servers is simply returning placeholder text, they will start according to the content specified in the configuration file. The configuration can be updated to achieve different results in subsequent executions. Changes include not starting both servers (specifying *false* for the start element of a server), changing port numbers, or adding additional servers, such as two web server instances.

Supporting Programs

The programs defined in the configuration file (*webserver* and *dataserver*) are minimal web servers that will display a message, but provide no other function. The port number that they use is defined in the configuration file and passed through the *–PORT* argument to the program. The source code for the *webserver* program (*Javascript/Node.js* and *Python*) are shown next. The *dataserver* program source is not shown (it is the same except for the message returned from the *HTTP GET*), but it is included in the accompanying materials.

Directory: chapter5, file: webserver.js

```
/*
 * Web Server
 */
var http = require ("http");

// port number to listen on for requests
```

```
var port = 8301;

// process arguments for port number argument
var command = process.argv.slice (2);
command.forEach (function (arg) {
    if (arg[0] === "-") {
        var elements = arg.split ("=");
        var key = elements[0].toUpperCase ();
        if ((key === "-P") || (key === "--PORT")) {
            port = elements[1];
        }
    }
});

// start HTTP server listener
var server = http.createServer (function (request, response) {
    // processing logic goes here
    response.writeHead (200, { "Content-type":"text/html" });
    response.write ("Web content goes here.");
    response.end ();
});

// listen for messages on specified port
server.listen (port);
console.log ("Web server listening on port " + port);
```

Directory: chapter5, file: webserver.py

```
"""
Web Server
"""
try:
    # Python 3
    from http.server import BaseHTTPRequestHandler, HTTPServer
except ImportError:
    # Python 2
    from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
import sys

def main ():
    # port number to listen on for requests
    port = 8301

    # process arguments for port number argument
    for arg in sys.argv[1:]:
        index = arg.find ("=")
        if index > -1:
            key = arg[0:index].upper ()
            value = arg[index + 1:len (arg)]
            if (key == "-P") or (key == "--PORT"):
                port = int (value)

    # listen for messages on specified port
    server = HTTPServer (("localhost", port), Handler)
    print ("Web server listening on port " + str (port))
    try:
        server.serve_forever ()
    except KeyboardInterrupt:
        server.shutdown ()
        server.server_close ()

class Handler (BaseHTTPRequestHandler):
    # processing logic goes here
    def do_GET (self):
        self.send_response (200)
```

```
        self.send_header ("Content-type", "text/html")
        self.end_headers ()
        self.wfile.write ("Web content goes here.".encode ("utf8"))
        return

if __name__ == "__main__":
    main ()
```

After the *startup* program has launched these servers, a web browser can be started and the servers can be interacted with using the *URLs*,

- *http://localhost:8301*
- *http://localhost:8302*

Update the port number to match the port specified in the configuration file if the original configuration is changed.

Summary

Configuration files can have a wide span of requirements. They often expand in scope over time, and often require the ability to express relationships between elements. As you read the next chapter, consider configuration cases where the relationships expressed in the organization / employee example can be applied to configuration scenarios.

6. Simple Data Management

There are a variety of options available for storing and accessing *JSON* content in databases. As seen in the prior examples, using files for persistent storage is also suitable for many uses. For programs that store persistent data, but have very limited data management needs, a text file using the *JSON* format is an option to consider. Some characteristics to factor into the decision,

- Is there an advantage to being able to view the data without any special tools (for instance, using a text editor)?
- Is the data intended to be editable, and if so, is use of a text editor an acceptable tool for editing?
- Is the data able to be read and written as a whole, rather than updates of individual objects/records?
- Is the data accessed only by a single person at a time?

If the answer to these characteristics is yes, or not applicable, then using *JSON* files as the basis for storing this data is worth considering.

If the answer to one, or both of, the last two questions is no, then *JSON* may still be applicable as a data format for content, but with a more robust data management system. Databases that support *JSON* content are available, and range in capability from basic data storage up to highly available distributed databases.

Usage Examples

Personal use applications such as note-taking, games, and address books can have limited data sets with varying content between stored objects.

Evaluation and educational versions of software programs may have limited use editions, with predefined data. These may also be intended for users that have limited familiarity with the program or use of database products, or have limited time to setup, configure, and manage a database. For these limited editions, a built-in, file based, data management function can be a suitable option.

Capabilities for Simple Data Management

Many of the capabilities for simple data management build on the those introduced by the configuration file example. Some additional capabilities are introduced when more general purpose data file uses are considered, including,

- Multiple files are more common with data files. This is partly historical, many file based programs have used one record format per file, since their flexibility was limited by positional data layouts (e.g., comma separated value files) or modeled after relational or similar record based data definition approaches.
- Use of keys and key references are often used to tie together records for different data elements. For example, a set of orders contains a customer number element that corresponds to a unique record in the set of customers.

Using *JSON* and *JSON Schema* is suitable for both single file and multiple file scenarios. When multiple files are present, and they are small, having the flexibility to consolidate them into a single file can be beneficial.

The use of keys and key references (e.g., foreign keys in relational databases) is not directly addressed by *JSON Schema* in its current draft. A couple of features can be used to address a portion of this capability.

- Constraints can be defined on the elements in the different definitions to ensure that keys and references are the same data type, and have the same content constraints (e.g., all customer numbers are integers in the range of 1 to 99999).
- The *required* constraint can be used to ensure a value is always present for those elements that will be referenced.

However,

- *JSON Schema* definitions do not include a definition mechanism to link two elements together. Therefore, knowledge of the relationship between the elements is left to the schema processor to implement independent of the schema definition.
- The check to determine whether the linked elements all have matching values is not part of the schema processing. To perform this validation, the schema processor needs to implement an additional feature to process these links to scan for lacking of matching elements.
- No capability is addressed for changes in the in-memory representation of the data or in the writing of the data that determines whether content being produced is valid (e.g., if a customer element is deleted, are any order elements containing references to that customer left orphaned).

If these capabilities are required, the program is required to supplement the *JSON Schema* capabilities to implement these features.

Example: Organization and Employee Data

The data model for employees in an organization provides a good example for covering the concepts related to *JSON Schema* use for simple data management. The benefits of this choice for the example are,

- Multiple files are used, and multiple object types are included in the organization file.
- Use of references, including a self reference (organizational hierarchy) and cross references (employee to organizational unit).
- The data model is familiar, and easy to extend.

To start, the organization data is shown.

Directory: chapter6, file: orgValid.json

```
{
  "units":
  [
    {
      "unitId":1,
      "unitOf":0,
      "name":"Corporate"
    },
    {
      "unitId":100,
```

```
        "unitOf":1,
        "name":"Finance"
      },
      {
        "unitId":101,
        "unitOf":100,
        "name":"Audit"
      },
      {
        "unitId":200,
        "unitOf":1,
        "name":"Human Resources"
      }
    ],
    "board":
    [
      {
        "name":"Ann Allen",
        "independent":true,
        "chair":false
      },
      {
        "name":"Bob Baker",
        "independent":true,
        "chair":false
      },
      {
        "name":"Carrie Conner",
        "independent":false,
        "chair":true
      }
    ]
  }
```

The organizational units are defined in a hierarchical structure, where a unit is either the top level unit (has no parent unit, indicated by the value 0 for the *unitOf* element) or a sub-unit (has a parent unit specified in the *unitOf* element). There is only one top level unit per organization.

A second object type is also included, the board members. Inclusion of these two related items with different data definitions in the same file, demonstrates the flexibility in placement of data items into persistent storage as suited to the individual requirements for each system.

Next, the employee data is shown.

File: chapter6, file: employeeValid.json

```
{
  "employees":
  [
    {
      "name":"Adam Ames",
      "unit":100,
      "title":"Staff Accountant"
    },
    {
      "name":"Barbara Barnes",
      "unit":101,
      "title":"Auditor"
    },
    {
      "name":"Carrie Conner",
      "unit":1,
      "title":"Board Chair and CEO"
    },
```

```
    {
        "name":"Dan Davis",
        "unit":200,
        "title":"Benefits Analyst"
    }
  ]
}
```

The employee data contains the name and title for each employee, along with the unit they work in. This unit corresponds with the organizational unit definition (*employee.unit* with *org.units.unitId*).

The schema definitions for the organization data follows.

File: chapter6, file: org_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Organization",
  "description":"Unit hierarchy and board members",

  "type":"object",
  "properties":
  {
    "units":
    {
      "type":"array",
      "items":
      {
        "type":"object",
        "properties":
        {
          "unitId":{"type":"integer", "minimum":1, "maximum":9999},
          "unitOf":{"type":"integer", "minimum":0, "maximum":9999},
          "name":{"type":"string"}
        },
        "additionalProperties":false,
        "required":["unitId", "unitOf", "name"]
      }
    },
    "board":
    {
      "type":"array",
      "items":
      {
        "type":"object",
        "properties":
        {
          "name":{"type":"string"},
          "independent":{"type":"boolean"},
          "chair":{"type":"boolean"}
        },
        "additionalProperties":false,
        "required":["name", "independent", "chair"]
      }
    }
  }
}
```

The schema definition for the employee data follows.

File: chapter6, file: employee_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Employee",
  "description":"Employee data",
```

```
  "type":"object",
  "properties":
  {
    "employees":
    {
      "type":"array",
      "items":
      {
        "type":"object",
        "properties":
        {
          "name":{"type":"string"},
          "unit":{"type":"integer", "minimum":1, "maximum":9999},
          "title":{"type":"string"}
        },
        "additionalProperties":false,
        "required":["name", "unit", "title"]
      }
    }
  }
}
```

Using the schema validation tool will show the data conforms to the schema definition for each, using the following commands.

```
validate orgValid.json org_schema.json
validate employeeValid.json employee_schema.json
```

In both cases, the result of the *validate* program will be a message showing the content is valid.

This example shows the applicability of *JSON* and *JSON Schema* to define the structure for persistent data, ability to model the data structures and content, and flexibility in implementation.

Valid Data, Invalid Cross-Reference

What happens when an employee is added, the assigned unit for the employee is in the valid range of unit numbers, but the unit does not exist in the organization data?

Directory: chapter6, file: employeeInvalid.json

```
{
  "employees":
  [
    {
      "name":"Adam Ames",
      "unit":100,
      "title":"Staff Accountant"
    },
    {
      "name":"Barbara Barnes",
      "unit":101,
      "title":"Auditor"
    },
    {
      "name":"Carrie Conner",
      "unit":1,
      "title":"Board Chair and CEO"
    },
    {
        "name":"Dan Davis",
        "unit":200,
        "title":"Benefits Analyst"
    },
```

```
      {
          "name":"Ed Edwards",
          "unit":300,
          "title":"Systems Analyst"

      }
    ]
  }
```

The new employee, Ed Edwards, is part of the Information Technology department (unit 300), however this unit is not yet included in the organization data. Run the validation program against the employee schema using,

```
validate employeeInvalid.json employee_schema.json
```

The result of the *validate* program is a message stating the content is valid. The unit number is validated against the schema definition of the number range between 1 and 99999, into which 300 is valid.

One option would be to update the employee schema definition to use an enumeration for the unit values, and to source the list of unit numbers from the org schema. For very static data this might be an appropriate choice, but for data that can change this is unlikely to be practical.

Another option is adding a validation step that supplements the base validation. Instead of using the validation tool as a standalone program, it will instead be used as a library and paired with custom logic to perform the cross-referencing portion of the validation.

Additional Custom Validation

In addition to the employee / organization unit cross reference, there are additional validation items that can be considered for inclusion in custom validation.

- Validate presence of one, and only one, element value across a set of elements (organization has only one top level unit).
- Verifies uniqueness of sub-elements across the set of elements (organizational *unitId*).
- Verifies the self references are valid for the organization hierarchy (*unitOf* always has a corresponding *unitId*).

JSON Schema supports uniqueness checking for schema elements, but this does not extend to discrete values within schema elements. For example, in *chapter 3* under *Array Constraints*, the *Validate (Duplicate Items)* example showed where two identical server definitions were rejected by the validation process since they did not conform with the *uniqueItems* constraint. However, the *uniqueItems* constraint could not be specified as only applying to a subset of the *server* object definition. In the case of the organization unit id element, the uniqueness validation is of interest to these discrete sub-elements, not the whole element.

The self-reference validation is a variant of the cross reference validation.

Custom Validation Processor

To provide the set of validation processing capabilities that include the base *JSON Schema* validation, along with the three additional custom validation steps identified, a custom validation processor will be created.

The program performs the following steps,

- Use the existing validate program and *JSON Schema* library to perform the syntax and schema validation processing of the content against the schema.
- Add new validation steps,
  - Verify that all *unitId* values are unique.
  - Logic to determine hierarchy validation for the organization. Verifies presence of one, and only one, top level unit. Verifies all *unitOf* references are to an existing unitId.
  - Logic to determine if all employee unit references are to an existing *unitId* in organization.

At the completion of the processing, a message is displayed indicating successful validation or an error message if the validation is not successful.

Launcher Module

A launcher module, *checkOrg.js* / *checkOrg.py*, is provided that processes the command line and calls the custom validation function.

Javascript / Node.js version

Directory chapter6, file: checkOrg.js

```
/**
 * Validate organization
 *
 * Usage: node check orgFile employeeFile
 */
var validateOrg = require ("./validateOrg").validateOrg;

// if module invoked directly, call the module function
if (require.main === module) {
    main ();
}

/**
 * Parse command line and initiate validation.
 */
function main () {
    var orgFile = null;
    var empFile = null;
    // process positional command line arguments
    var args = process.argv.slice (2);
    args.forEach (function (arg) {
        if (arg[0] !== "-") {
            // assign positional arguments
            if (orgFile === null) {
                orgFile = arg;
            } else {
                empFile = arg;
            }
        }
    });

    // if both files not specified, command is invalid
    if ((orgFile === null) || (empFile === null)) {
        console.log ("Usage: node check orgFile employeeFile");
        console.log ("  orgFile       JSON file - organization");
        console.log ("  employeeFile  JSON file - employees");
        process.exit (1);
    }
```

```
    // call organization validation processor
    validateOrg (orgFile, empFile);
}

// module exports
exports.main = main;
```

Python version

Directory: chapter6, file: checkOrg.py

```
"""
Validate organization

Usage: python check.py orgFile employeeFile
"""
from argparse import ArgumentParser
import sys
from validateOrg import validateOrg

def main ():
    """
    Parse command line and initiate validation.
    """
    parser = ArgumentParser ()
    parser.add_argument ("orgFile", help="Organization input file")
    parser.add_argument ("empFile", help="Employee input file")
    args = parser.parse_args ()

    # call organization validation processor
    validateOrg (args.orgFile, args.empFile)

if __name__ == "__main__":
    main ()
```

The launcher calls the custom validation processor with the names of the organization and employee data files to validate.

Custom Validation Processor – Javascript / Node.js Version

Directory: chapter6, file: validateOrg.js

The first part includes the callable function, *validateOrg*. It loads the organization and employee schemas, and runs the base validation processor against the provided files respectively. It then invokes each of the custom validation functions.

```
/**
 * Validate the organization data and employee data
 */
var jsonvalidate = require ("ujs-jsonvalidate");
var validate = jsonvalidate.validate;

/**
 * Validate organization structure and content.
 * @param {string} orgFile Organization JSON file name.
 * @param {string} empFile Employee JSON file name.
 */
function validateOrg (orgFile, empFile) {
    var orgSchema = "org_schema.json";
    var empSchema = "employee_schema.json";

    // validate organization data
    var units = null;
    validate (orgFile, orgSchema, null, null, function (code, data, msg) {
```

```
        if (code === jsonvalidate.VALID) {
            units = data.units;
        } else {
            console.log ("Error processing organization: " + msg);
            process.exit (code);
        }
    });

    // validate employee data
    var employees = null;
    validate (empFile, empSchema, null, null, function (code, data, msg) {
        if (code === jsonvalidate.VALID) {
            employees = data.employees;
        } else {
            console.log ("Error processing employees: " + msg);
            process.exit (code);
        }
    });

    // call custom validation functions
    verifyTopLevelUnit (units);
    verifyUniqueUnitIds (units);
    verifyUnitIds (units);
    verifyEmployeeUnits (employees, units);
}
```

The first custom validation function, *verifyTopLevelUnit*, determines whether the organization contains the correct top level unit structure. It does this by inspecting the units, and determining whether there is one, and only one, unit that has no parent unit. A success or failure message is displayed.

```
/**
 * Verify org has one, and only one, top level unit.
 * @param {object[]} units Array of unit objects.
 */
function verifyTopLevelUnit (units) {
    var topLevelUnitsCount = 0;
    for (var ctr = 0; ctr < units.length; ctr ++) {
        if (units[ctr].unitOf === 0) {
            topLevelUnitsCount ++;
        }
    }
    // display results
    if (topLevelUnitsCount === 0) {
        console.log ("Error: Missing top level unit");
    } else if (topLevelUnitsCount === 1) {
        console.log ("Valid: Organization top level unit valid");
    } else {
        console.log ("Error: Multiple top level units defined");
    }
}
```

The second custom validation function, *verifyUniqueUnitIds*, checks for duplicate unit identifiers. A message is displayed for any duplicates found.

```
/**
 * Verify unitId is unique across all units.
 * @param {object[]} units Array of unit objects.
 */
function verifyUniqueUnitIds (units) {
    for (var ctr1 = 0; ctr1 < units.length; ctr1 ++) {
        var currentUnitId = units[ctr1].unitId;
        for (var ctr2 = ctr1 + 1; ctr2 < units.length; ctr2 ++) {
            if (currentUnitId === units[ctr2].unitId) {
                console.log ("Error: Duplicate unitId " + currentUnitId);
```

```
                break;
            }
        }
    }
}
```

The third custom validation function, *verifyUnitIds*, verifies that all units in the organization have a parent unit. Failure messages are displayed for any orphan units. A success message is displayed if the hierarchy has no orphan units.

```
/**
 * Verify org has valid unitId for all unitOf references.
 * @param {object[]} units Array of unit objects.
 */
function verifyUnitIds (units) {
    var orgValid = true;
    for (var ctr1 = 0; ctr1 < units.length; ctr1 ++) {
        var unitOf = units[ctr1].unitOf;
        if (unitOf !== 0) {
            var validUnitOf = false;
            for (var ctr2 = 0; ctr2 < units.length; ctr2 ++) {
                if (unitOf === units[ctr2].unitId) {
                    validUnitOf = true;
                    break;
                }
            }
            if (validUnitOf === false) {
                console.log ("Error: Invalid unitOf " + unitOf);
                orgValid = false;
            }
        }
    }
    if (orgValid === true) {
        console.log ("Valid: Organization hierarchy is valid");
    }
}
```

The last custom validation function, *verifyEmployeeUnits*, works across the two data sets, determining whether all employees have valid unit identifiers by verifying the unit identifiers are valid in the organization data. A failure message is displayed for any employee assigned to an invalid unit. A success message is displayed if all employees are assigned to valid units.

```
/**
 * Verify all employee unit references are valid org units.
 * @param {object[]} employees Array of employee objects.
 * @param {object[]} units Array of unit objects.
 */
function verifyEmployeeUnits (employees, units) {
    var allValid = true;
    for (var ctr1 = 0; ctr1 < employees.length; ctr1 ++) {
        var validUnit = false;
        var empUnit = employees[ctr1].unit;
        for (var ctr2 = 0; ctr2 < units.length; ctr2 ++) {
            if (empUnit === units[ctr2].unitId) {
                validUnit = true;
                break;
            }
        }
        if (validUnit === false) {
            console.log ("Error: Invalid employee unit " + empUnit);
            allValid = false;
        }
    }
```

```
    if (allValid === true) {
        console.log ("Valid: All employee unit references valid");
    }
}

// exports
exports.validateOrg = validateOrg;
```

This last section also declares the export for the *validateOrg* function. The execution instructions to exercise the custom validation tests follow the *Python* section.

Custom Validation Processor – Python Version

Directory: chapter6, file: validateOrg.py

The first part includes the callable function, *validate*. It loads the organization and employee schemas, and runs the base validation processor against the provided files respectively. It then invokes each of the custom validation functions.

```
"""
 * Validate the organization data and employee data
"""
from jsonValidate import validate
import sys

def validateOrg (orgFile, empFile):
    """
    Validate organization structure and content.
    Args:
        orgFile Organization JSON file name.
        empFile Employee JSON file name.
    """
    orgSchema = "org_schema.json"
    empSchema = "employee_schema.json"

    # validate organization data
    units = None
    code, data, message = validate (orgFile, orgSchema, None, None)
    if code == 0:
        units = data["units"]
    else:
        print ("Error processing organization: " + message)
        sys.exit (code)

    # validate employee data
    employees = None
    code, data, message = validate (empFile, empSchema, None, None)
    if code == 0:
        employees = data["employees"]
    else:
        print ("Error processing employees: " + message)
        sys.exit (code)

    verifyTopLevelUnit (units)
    verifyUniqueUnitIds (units)
    verifyUnitIds (units)
    verifyEmployeeUnits (employees, units)
```

The first custom validation function, *verifyTopLevelUnit*, determines whether the organization contains the correct top level unit structure. It does this by inspecting the units, and determining whether there is one, and only one, unit that has no parent unit. A success or failure message is displayed.

```
def verifyTopLevelUnit (units):
```

```
    """ verify org has one, and only one, top level unit """
    topLevelUnitsCount = 0
    for unit in units:
        if unit["unitOf"] == 0:
            topLevelUnitsCount += 1

    # display results
    if topLevelUnitsCount == 0:
        print ("Error: Missing top level unit")
    elif topLevelUnitsCount == 1:
        print ("Valid: Organization top level unit valid”)
    else:
        print ("Error: Multiple top level units defined")
```

The second custom validation function, *verifyUniqueUnitIds*, checks for duplicate unit identifiers. A message is displayed for any duplicates found.

```
def verifyUniqueUnitIds (units):
    """ verify unitId is unique across all units """
    for index1 in range (len (units)):
        currentUnitId = units[index1]["unitId"]
        for index2 in range (index1 + 1, len (units)):
            if currentUnitId == units[index2]["unitId"]:
                print ("Error: Duplicate unitId " + str (currentUnitId))
                break
```

The third custom validation function, *verifyUnitIds*, verifies that all units in the organization have a parent unit. Failure messages are displayed for any orphan units. A success message is displayed if the hierarchy has no orphan units.

```
def verifyUnitIds (units):
    """ verify org has valid unitId for all unitOf references """
    orgValid = True
    for units1 in units:
        if units1["unitOf"] != 0:
            validUnitOf = False
            for units2 in units:
                if units1["unitOf"] == units2["unitId"]:
                    validUnitOf = True
                    break

            if not validUnitOf:
                print ("Error: Invalid unitOf " + str (units1["unitOf"]))
                orgValid = False

    if orgValid:
        print ("Valid: Organization hierarchy is valid")
```

The last custom validation function, *verifyEmployeeUnits*, works across the two data sets, determining whether all employees have valid unit identifiers by verifying the unit identifiers are valid in the organization data. A failure message is displayed for any employee assigned to an invalid unit. A success message is displayed if all employees are assigned to valid units.

```
def verifyEmployeeUnits (employees, units):
    """ verify all employee unit references are valid org units """
    allValid = True
    for employee in employees:
        validUnit = False
        empUnit = employee["unit"]
        for unit in units:
            if empUnit == unit["unitId"]:
                validUnit = True
                break
```

```
        if not validUnit:
            print ("Error: Invalid employee unit " + str (empUnit))
            allValid = False

    if allValid:
        print ("Valid: All employee unit references valid")
```

To exercise the custom validation functions, the next step executes the custom validation processor with different data sets.

Validation Data Examples

A variety of organization and employee data files are provided, allowing combinations to be used to exercise each of the functions provided in the custom validation processor.

Validation (No Errors)

First, the new validation program will be run with the valid employee data. This verifies that the new program is consistent with the base validation in recognizing correct content. Use one of the following commands.

```
node checkOrg.js orgValid.json employeeValid.json
python checkOrg.py orgValid.json employeeValid.json
```

The result of the command will be a valid content message.

In the next examples, the validation program is run with the invalid content that passes the *JSON Schema* base validation (correct structure and content), but contains incorrect data that is outside the bounds of the *JSON Schema* specification.

Validation (Missing Top Level Unit)

If the top level unit of the organization is removed (the unit with *unitOf* value 0), then the organization content will be considered invalid. Use one of the following commands.

```
node checkOrg.js orgInvalid1.json employeeValid.json org
python checkOrg.py orgInvalid1.json employeeValid.json org
```

The result will indicate that the content is invalid with message *"Error: Missing top level unit in organization"*.

Validation (Multiple Top Level Units)

If more than one top level unit of the organization is present (multiple units with *unitOf* value 0), then the organization content will be considered invalid. Use one of the following commands.

```
node checkOrg.js orgInvalid2.json employeeValid.json org
python checkOrg.py orgInvalid2.json employeeValid.json org
```

The result will indicate that the content is invalid with message *"Error: Multiple top level units in organization"*.

Validation (Non-Unique Sub-Element)

No two units can have the same unit identifier (*unitId*). If there are duplicates, the content is invalid. Use one of the following commands.

```
node checkOrg.js orgInvalid3.json employeeValid.json org
python checkOrg.py orgInvalid3.json employeeValid.json org
```

The message displayed will indicate that a unit identifier (101) in the organization is a duplicate.

Validation (Invalid Self Reference)

The organization data model uses a self referencing structure to relate each unit with its parent unit. If the parent of a unit (*unitOf*) is not valid, then the content is considered invalid. Use one of the following commands.

```
node checkOrg.js orgInvalid4.json employeeValid.json org
python checkOrg.py orgInvalid4.json employeeValid.json org
```

The message displayed will indicate that a self reference (2) is not to a valid unit in the organization.

Validation (Invalid Cross Reference)

Each employee is associated with a unit of the organization. When the unit in the employee element does not match a unit in the organization, the combination of the organization and employee content is invalid, even if each is individually valid otherwise. Use one of the following commands.

```
node checkOrg.js orgValid.json employeeInvalid.json org
python checkOrg.py orgValid.json employeeInvalid.json org
```

The message displayed will indicate that the employee unit (300) in the new employee is not a valid unit in the organization.

Persistent State Validation Versus In Flight Validation

So far, the validation processes have been defined and performed in the context of the persistent state of the content. In the initial definition for simple data management, allowing external editing of the data was one of the possible capabilities, which then requires the persistent state to be validated in whole.

However, this is not the only possible approach to using *JSON Schema* for database uses. As shown in the various examples, schema definitions can be very rich. From the message exchange example, the validation can be performed as content is received. Combining these two concepts, a validation model for changes to data can be derived. When a change to data is being performed (addition, update, or deletion) the schema definition can be used to determine whether the change will result in a valid data representation.

- Are the elements introduced complete and correct according to the schema definition?
- Does the resulting data representation after the changes maintain completeness and correctness?

With *JSON Schema*, the integrity within the scope of the element being changed is well covered. The data management engine receiving the change request can

- Validate the incoming content, using the schema processor
- Construct an interim representation of the potential change, placing the new/changed content in the context of the whole database.
- Validate the interim representation, for example ensuring that a *uniqueItems* constraint is met.
- When validation is complete, apply the changes to the database.

For additional custom validation steps, the in flight validation processor can provide these steps in a manner similar to the organization example in this chapter. The base *JSON Schema* validation provides the first level of validation, and then additional validation steps can be provided by the custom processor. These can include,

- Unique element checking, for example if an element is used as a key that requires it to be unique.

- Cross reference checking if the database supports a foreign key type of mechanism.

There are many potential variations on how content can be managed and accessed, but a common step across these alternatives is the core validation processing provided by a *JSON Schema* processor. The flexibility to build on, and extend, this capability to fit many use cases and technologies is a key recognition of the role the schema plays.

Growing Into a Database

Program requirements can change over time. The simple data management requirements for the first version of a program may grow into requirements that are no longer a good fit for a file based persistence approach. Fortunately, moving to a database persistence option is possible.

Generally, databases supporting *JSON* content do not build in general purpose or extensible schema validation capabilities. Some limited validation is done to ensure that content is valid, and that required elements are present (typically an identifier). However, the schema definition for the general content remains in the scope of the program, not the database.

- If storing the data as a whole doesn't change, then the *JSON* content stays the same. While the storage container switches from a file to a database entry, the content itself doesn't change.
- The relationships between the program, schema validation processor, schema validation extensions and the persistence function remain the same, although their implementations will reflect interactions with the substituted persistence function.
- The validation processing can apply both inbound (read) and outbound (create, update) interactions.

If the new requirements also drive a different storage approach for some, or all, of the data, then the content will remain the same for the elements being stored, but the structural elements may change. For instance,

- Splitting the unit hierarchy and board elements in the organization schema would be a simple change. The validation for each element can be applied independently for the retrieval / storage of each element.
- Treating each unit as an individual document would be a translation from a single array object being stored to a collection containing the set of unit documents. While not affecting the schema validation for the content of each unit definition, the custom validation logic that validates across the set of units would include logic to work with the database collection, rather than just retrieving with the single array element.

If the driving requirements are related to adding transactional capabilities, the transition of some elements to a more granular storage representation will be typical. However, if the requirements are driven by consolidation, then minimal changes are likely to be needed.

Domain Specific Validators

The custom validation logic shown in the employee and organization example is an application specific validator. However, each of the validation steps is likely to be usable in many more contexts, though the schemas and element names may differ for each.

This raises the opportunity for domain specific validators, where the validation logic is applied by using a template or generic validation function and feeding it schema specific content.

When considering the relationship between elements, there are two common scenarios that this may apply to.

- Type consistency. For example, the unit an employee is assigned in the employee data and the unit is also defined in the organization data. These two definitions must be consistent across the two places they are used.
- Value consistency. For example,
  - In the organization data, all organization unit identifiers must be unique.
  - In a cross element relationship, the unit an employee is assigned must be present as a unit in the organization data.

For the type consistency case, when both schema definitions are managed together, the use of references allows a single definition to be used for many element definitions. However, if the schema definitions are not managed together, or arbitrary schemas can be combined, then the domain specific validator can read each schema and determine whether the element definition pairs inspected are compatible.

When the type consistency is not an exact match, it can be augmented with value consistency. For example, one schema may specify unit as an integer in the range 0 to 1000, and the second may have the range 0-99999. The content validation can determine that all content is compliant to both (i.e., all content is valid for the union of the ranges).

The logic for value consistency between elements of the same schema is a generic pattern, which can be applied to many schemas. Whether the elements are contained in one schema or cross schema, defining the relationships and performing the validation are straightforward.

Using the organization and employee example, consider how the different validation steps could be applied with other schemas, and how they could be generalized to be used with any schema (template or configuration driven).

Other domains may apply special meaning to some content values, which may not be possible to express in *JSON Schema*. For example, when multiple elements are constrained by a formula rather than by individual value. The domain specific validator can apply the formula to the collective elements, such as *((Element A + Element B) * Element C < 1.0)*.

Augmenting the core *JSON Schema* validation processes with domain specific validation processes is a mechanism that can be used to extend the validation scope to include relationship validation and / or address domain specific requirements.

7. Designing Software for JSON Message Exchange

Message exchange often occurs between systems that are managed independently (whether separate organizations or different groups within an organization). In many cases, there will be many parties that interact. It is important to ensure that message content is correct and complete. Invalid messages can be generated through errors (program or human), mismatch in versions of a message format, or malevolence.

- Validation ensures the content of the message has the expected structure and content.
- Validation can apply in both directions. Not only can it be performed on incoming message content, it can be performed on outgoing messages to ensure messages sent are valid.
- In conjunction with other security capabilities, validation can be part of an overall system to detect and reject invalid traffic at an early point in its processing.

For some types of content, the content contains elements that are arbitrary. An example is a message that contains a web page, where the defined elements includes the site address, but the page content itself is arbitrary *HTML*. For this scenario, the schema definition can be used by the validation processing to determine if the required elements are present and correctly formatted, and that the arbitrary content is in the correct location. However, other processing, such as a malware scanner, may be required to ensure that the arbitrary content is allowable.

Implementing Programs that use JSON Message Exchange

Sending and receiving messages that contain *JSON* content is very common for programs interacting through the Internet or intranets. *RESTful* services often send and receive *JSON* content. Some web platforms and libraries include built-in support for *JSON* content. This includes recognizing the *application/json* media type and providing library support for serializing / deserializing and receiving / creating messages with this media type.

For a server program supporting *JSON* content, the program logic at a high level will,

- Open a channel to receive messages at. In the case of a *RESTful* server, this will be an *HTTP* or *HTTPS* port.
- While running, performing the following steps for each message received,
  - Accept messages received on the channel.
  - Validate the message received. Process the message in the case of a valid message, or return an error in the case of an invalid message.
  - Preparing a response message based on the processing performed.
  - Optionally, validate the response message.
  - Send the response message.
- When the program is finished, or directed to terminate, close the channel.

The example is a program that accepts a message with a pair of digits to add together and returns the result in a message. If the received message is not valid, an error will be returned instead. The schema for the request message follows.

Directory: chapter7, file: addRequest_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Addition request",
  "description":"Numbers to add together",

  "type":"object",
  "properties":
  {
    "number1":{"type":"integer"},
    "number2":{"type":"integer"}
  },
  "additionalProperties":false,
  "required":["number1", "number2"]
}
```

The schema for a response message that contains a valid answer follows.

Directory: chapter7, file addResponse_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Addition response",
  "description":"Answer from addition service",

  "type":"object",
  "properties":
  {
    "answer":{"type":"integer", "minimum":1 }
  },
  "additionalProperties":false,
  "required":["answer"]
}
```

Note that the *answer* property has a minimum value of 1. This allows testing for an invalid response in the client program by passing the addition service two zeros to add.

The schema for a response message that contains an error follows.

Directory: chapter7, file addError_schema.json

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "title":"Addition error",
  "description":"Error message from addition service",

  "type":"object",
  "properties":
  {
    "error":{"type":"string"}
  },
  "additionalProperties":false,
  "required":["error"]
}
```

Examples of valid and invalid content are included in the *additionClient* program). Content for valid data message (contained in *additionClient* program).

```
{
  "number1":15,
  "number2":24
}
```

Content for invalid data message (contained in *additionClient* program). The message is invalid since the value *false* is not acceptable for the property *number2*.

```
{
  "number1":15,
  "number2":false
}
```

To test the client schema validation of a response, a request sending two zero values is sent. This is a valid input for the addition service, but the result is not valid according to the response schema. Since the addition service does not do an outbound validation, this invalid content is not discovered until the client performs the validation step.

```
{
  "number1":0,
  "number2":0
}
```

Since the context for the example is a message exchange, rather than the content being present on a persistent storage medium, the validation will occur in program logic for the server rather than using the *validate* program.

Javascript / Node.js Implementation

The server and client implemented with *Javascript / Node.js*.

Javascript / Node.js Server

In the *Javascript / Node.js* version of the server, the *Tiny Validator* is used as the schema validation processor.

Directory: chapter7, file: additionService.js

The leading content, defines two module variables, *port*, for the port number the server will listen on and *requestSchema*, for the schema that is used to validate received messages.

```
 1 /**
 2  * Addition service using JSON and JSON Schema.
 3  *
 4  * Starts an HTTP server listening for addition requests.
 5  * Server default port is 8303.
 6  */
 7 var fs = require ("fs");
 8 var http = require ("http");
 9 var tv4 = require ("tv4");
10
11 var port = 8303;
12 var requestSchema = null;
13
14 // if module invoked directly, call main
15 if (require.main === module) {
16     main ();
17 }
```

The *main* function calls the command line processor, loads the schema to be used to validate incoming messages and then starts the server. At the end, it displays a message indicating the port number being listened on.

```
19 /**
20  * Program entry point.
21  */
22 function main () {
23     // process command line for port number
24     processCommand ();
25
```

```
26     // Load JSON Schema to validate result against
27     try {
28         var data = fs.readFileSync ("addRequest_schema.json");
29         requestSchema = JSON.parse (data);
30     } catch (e) {
31         console.log ("Error loading request schema: " + e.message);
32         process.exit (1);
33     }
34
35     // listen for messages on specified port
36     var server = http.createServer (handler);
37     server.listen (port);
38     console.log ("Addition service listening on port " + port);
39 }
```

The *handler* function is invoked every time an *HTTP* message is received on the listened to port. When a message is received, a console message will be displayed. The content type will be checked to verify it is the *JSON* content type. If so, the *body* variable will be populated with the message content through the request.*on ("data", ...)* function. When the request has been fully received, the request.*on ("end", …)* will call the addition function to process the message.

```
41 /**
42  * HTTP request handler.
43  * @param request HTTP request object.
44  * @param response HTTP response object.
45  */
46 function handler (request, response) {
47     // when a message is received, display a message
48     console.log ("Request received");
49
50     // verify the content type is for JSON content
51     var contentType = request.headers["content-type"];
52     if (contentType !== "application/json") {
53         console.log ("Invalid content type: " + contentType);
54     } else {
55         // initialize request content with empty string
56         var body = "";
57
58         // when data is received, add it to request content
59         request.on ("data", function onData (data) {
60             body += data;
61         });
62
63         // when all data is received, process the content
64         request.on ("end", function onEnd () {
65             addition (response, body);
66         });
67     }
68 }
```

The addition function processes the message through the following steps,

- Parses the body text into a *JSON* object.
- Initializes the values to be used in the response message.
- Creates a fresh validation processor instance, and validates the received message content against the requestSchema loaded earlier.
- If the content is valid, the return value is calculated and populated into the result.
- If the content is not valid, the error content is populated into the result.

- The response message is created and sent. Note, the result object is converted to a text representation for the transport.

```
 70 /**
 71  * Process the addition request.
 72  * @param response HTTP response object
 73  * @param body HTTP body text
 74  */
 75 function addition (response, body) {
 76     console.log ("addition body = " + body);
 77     // display received content and parse to JSON object
 78     var input = JSON.parse (body);
 79
 80     var result = null;
 81     var contentType = { "Content-type": "application/json" };
 82
 83     // validate against schema
 84     var validator = tv4.freshApi ();
 85     if (validator.validate (input, requestSchema) === true) {
 86         // calculate result and store in JSON response object
 87         result = { "answer": input.number1 + input.number2 };
 88         response.writeHead (200, contentType);
 89     } else {
 90         result = { "error": "invalid request"};
 91         response.writeHead (400, contentType);
 92     }
 93
 94     response.write (JSON.stringify (result));
 95     response.end ();
 96 }
```

The *processCommand* function sets the port number if a command line argument has been specified to set it.

```
 98 /**
 99  * Set port from command line arguments.
100  */
101 function processCommand () {
102     var command = process.argv.slice (2);
103     command.forEach (function (arg) {
104         if (arg[0] === "-") {
105             var elements = arg.split ("=");
106             var key = elements[0].toUpperCase ();
107             if ((key === "-P") || (key === "--PORT")) {
108                 port = elements[1];
109             }
110         }
111     });
112 }
```

Node.js, provides the *HTTP* serving functions in its base library. In the *main* function, the *server.listen* call will initiate the event loop that will keep running until the program is terminated.

Javascript / Node.js Client

The client program will generates messages towards the server. The messages, described previously, follow different paths through the client code.

The source code for the client follows.

Directory: chapter7, file additionClient.js

The leading content defines the module variables. The port number, *port*, the client will send requests on, and the two schemas to validate the responses – *resultSchema* and *errorSchema*.

```
 1 /**
 2  * Client to the addition service using JSON and JSON Schema.
 3  *
 4  * HTTP client to make requests. Default port is 8303.
 5  */
 6 var fs = require ("fs");
 7 var http = require ("http");
 8 var tv4 = require ("tv4");
 9
10 var port = 8303;
11 var responseSchema = null;
12 var errorSchema = null;
13
14 // if module invoked directly, call main
15 if (require.main === module) {
16     main ();
17 }
```

The *main* function calls the command line processor, allowing the port number option to be used. It then loads the two schemas, and calls the *makeRequests* function to generate the requests.

```
19 /**
20  * Program entry point.
21  */
22 function main () {
23     // process command line for port number
24     processCommand ();
25
26     // Load JSON Schema to validate result against
27     try {
28         var data = fs.readFileSync ("addResponse_schema.json");
29         responseSchema = JSON.parse (data);
30         data = fs.readFileSync ("addError_schema.json");
31         errorSchema = JSON.parse (data);
32     } catch (e) {
33         console.log ("Error loading result schemas: " + e.message);
34         process.exit (1);
35     }
36
37     makeRequests ();
38 }
```

The *makeRequests* function generates the requests shown above, one valid, one with invalid content, and one that will generate an invalid result. For each request, the *JSON* content is created and *postRequest* is called.

```
40 /**
41  * Make requests with valid and invalid content.
42  */
43 function makeRequests () {
44     // make a request with valid content
45     var input = { "number1": 15, "number2": 24 };
46     var validRequest = JSON.stringify (input);
47     postRequest ("Add 2 numbers", validRequest);
48
49     // make a request with invalid content
50     input = { "number1": 15, "number2": true };
51     var invalidRequest = JSON.stringify (input);
52     postRequest ("Add number and boolean", invalidRequest);
53
54     // make a request that will get an invalid result
55     input = { "number1": 0, "number2": 0 };
56     var invalidResult = JSON.stringify (input);
57     postRequest ("Add two zeros", invalidResult);
```

```
 58 }
```

The *postRequest* function performs the following steps,

- Sets up the request details including headers and addressing (lines 67-77).
- Creates the callback to receive the message response, in the context of the last parameter on line 80. The callback verifies the content is the correct content type (line 83), accumulates the message fragments (lines 87-92), and when the message is fully received calls the *processResult* function (lines 95-97).
- The *HTTP* request is generated and sent (lines 102-103).

```
 60 /**
 61  * Post a request to the additionService.
 62  * @param name Request name to display with result
 63  * @param content JSON object to pass to additionService
 64  */
 65 function postRequest (name, content) {
 66     // create request definition
 67     var headers = {
 68         "Content-type": "application/json",
 69         "Content-length": content.length
 70     };
 71     var options = {
 72         "host": "localhost",
 73         "port": port,
 74         "path": "/",
 75         "method": "POST",
 76         "headers": headers
 77     };
 78
 79     // create request, and accept response messages
 80     var request = http.request (options, function (response) {
 81         // verify the content type is for JSON content
 82         var contentType = response.headers["content-type"];
 83         if (contentType !== "application/json") {
 84             console.log ("Invalid content type: " + contentType);
 85         } else {
 86             // initialize response content with empty string
 87             var body = "";
 88
 89             // when data is received, add it to response content
 90             response.on ("data", function onData (data) {
 91                 body += data;
 92             });
 93
 94             // when all data is received, process the content
 95             response.on ("end", function onEnd () {
 96                 processResult (name, response, body);
 97             });
 98         }
 99     });
100
101     // place the content in the body and send the request
102     request.write (content);
103     request.end ();
104 }
```

The *processResult* function displays the result of the request. The first line displayed is the request information. A fresh instance of the validation processor is created (line 115). The message body received is parsed (line 118), and based on the status code in the response message, the body is validated. If the response status was 200 (*OK*), then the result is validated against the

addResponse_schema.json schema (lines 121-127). If the response status code was 500 (*BAD_REQUEST*), then the result is validated against the *addError_schema.json* schema (lines 128-134).

If a valid result was received, then the addition result is displayed. Otherwise the error result or the validation error are displayed.

```
106 /**
107  * Process the response message.
108  * @param name Name of the request.
109  * @param response HTTP response object.
110  * @param data HTTP body text.
111  */
112 function processResult (name, response, body) {
113     console.log ("\nResult for request: " + name);
114     var error = null;
115     var v = tv4.freshApi ();
116 
117     // parse response content to JSON object
118     var result = JSON.parse (body);
119 
120     // if response status was 200 (OK)
121     if (response.statusCode === 200) {
122         // validate against schema, if valid, display answer
123         if (v.validate (result, responseSchema) === true) {
124             console.log ("  Result = " + result.answer);
125         } else {
126             error = v.error;
127         }
128     } else if (response.statusCode === 400) {
129         // if response status was an error 400 (BAD REQUEST)
130         if (v.validate (result, errorSchema) === true) {
131             console.log ("  Server error: " + result.error);
132         } else {
133             error = v.error;
134         }
135     }
136 
137     // if error, print error details
138     if (error !== null) {
139         // display validation error
140         console.log ("  Data is not valid in the response");
141         console.log ("  Message: " + error.message);
142         console.log ("  Data path: " + error.dataPath);
143         console.log ("  Schema path: " + error.schemaPath);
144     }
145 }
```

The *processCommand* function sets the port number if a command line argument has been specified to set it.

```
147 /**
148  * Set port from command line arguments.
149  */
150 function processCommand () {
151     var command = process.argv.slice (2);
152     command.forEach (function (arg) {
153         if (arg[0] === "-") {
154             var elements = arg.split ("=");
155             var key = elements[0].toUpperCase ();
156             if ((key === "-P") || (key === "--PORT")) {
157                 port = elements[1];
158             }
159         }
160     });
```

```
161 }
```

After making the requests and displaying their results, the program will end.

Python Implementation

The server and client implemented with *Python*.

Python Server

In the *Python* version of the server, the *jsonschema* library is used as the schema validation processor.

Directory: chapter7, file: additionService.py

The leading content defines the imports and the *main* function. The *main* function creates the *AdditionService* instance, which initiates the server.

```
 1 """
 2 Addition service using JSON and JSON Schema.
 3
 4 Starts an HTTP server listening for addition requests.
 5 Server default port is 8303.
 6 """
 7 try:
 8     # Python 3
 9     from http.server import BaseHTTPRequestHandler, HTTPServer
10 except ImportError:
11     # Python 2
12     from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
13 from argparse import ArgumentParser
14 from json import loads, dumps
15 from jsonschema import Draft4Validator
16 import sys
17
18 def main ():
19     """ Program entry point. """
20     AdditionService ()
```

The *AdditionService* class is declared with the *init* method which sets the *port* (default or from a command line parameter), and creates the *HTTP* server instance.

```
22 class AdditionService:
23     """ Start server for addition service. """
24     def __init__ (self):
25         """ Set port and start server """
26         # process command line for port number
27         self.port = 8303
28         self.processCommand ()
29
30         # listen for messages on specified port
31         server = HTTPServer (("localhost", self.port), Handler)
32         print ("Addition service listening on port " + str (self.port))
33         try:
34             server.serve_forever ()
35         except KeyboardInterrupt:
36             server.shutdown ()
37             server.server_close()
```

The *processCommand* method checks the command line arguments for a port option.

```
39     def processCommand (self):
40         """ Get port from command line arguments. """
41         parser = ArgumentParser ()
42         parser.add_argument ("-p", "--port", type=int, dest="port",
43           action="store", help="Port to make requests on")
```

```
44          args = parser.parse_args ()
45          if args.port is not None:
46              self.port = args.port
```

The *Handler* class defines the methods that are invoked when messages are received. The only message being processed in this case is *HTTP POST*, which is handled by the *do_POST* method. The class starts with declaring a class variable, *requestSchema* (line 51) and the method, *loadRequestSchema* (lines 53-67) that loads the schema used to check the request.

```
48 class Handler (BaseHTTPRequestHandler):
49     """ HTTP request handler """
50     # class static, only load once
51     requestSchema = None
52
53     def loadRequestSchema (self):
54         """ load request schema from file (once only) """
55         # Load JSON Schema to validate input against
56         try:
57             # read the file and convert to a JSON object
58             data = open ("addRequest_schema.json", "rU").read ()
59         except IOError as e:
60             print ("Error loading input schema: " + e.strerror)
61             sys.exit (1)
62
63         try:
64             Handler.requestSchema = loads (data)
65         except Exception as e:
66             print ("Invalid JSON content in input schema")
67             sys.exit (1)
```

The *do_POST* method performs the following tasks,

- Writes a message to the console indicating a message was received (line 72).
- If the request schema is not yet loaded, loads it (lines 75-76).
- Verifies the content is the correct type (lines 78-80).
- Gets the message body, and parses the *JSON* content. (lines 82-85).
- Validates the request against the *addRequest_schema.json* schema (lines 90-91).
- If the request is valid, create the response content. (lines 94-98).
- If an error occurs, the error response content is created (lines 99-102).
- The response message is created (lines 104-106) with headers and the response content.

```
69     # web processing logic goes here
70     def do_POST (self):
71         """ process POST, generate response """
72         print ("Request received")
73
74         # if inputSchema not loaded, load once
75         if Handler.requestSchema is None:
76             Handler.loadRequestSchema (self)
77
78         contentType = self.headers["content-type"]
79         if contentType != "application/json":
80             print ("Invalid content type: " + contentType)
81         else:
82             length = int (self.headers["Content-Length"])
83             data = self.rfile.read (length).decode ("utf8")
84             print ("addition body = " + data)
85             dataIn = loads (data)
86
```

```
            #validate
            try:
                print ("start validate")
                validator = Draft4Validator (Handler.requestSchema)
                validator.validate (dataIn)
                print ("validated")

                answer = dataIn["number1"] + dataIn["number2"]
                print ("answer " + str (answer))
                result = dumps ({ "answer": answer })
                print ("result " + result)
                self.send_response (200)
            except Exception as e:
                # if validation failed, return error
                result = """{"error": "Invalid request"}"""
                self.send_response (400)

            self.send_header ("Content-type", "application/json")
            self.end_headers ()
            self.wfile.write (result.encode ("utf8"))
```

The ending lines handle invocation of the *main* function when the program is initiated.

```
if __name__ == "__main__":
    main ()
```

The program will run until interrupted (using *Ctrl-C*).

Python Client

The client program will generate messages towards the server. The messages, described previously, follow different paths through the client code.

The source code for the client follows.

Directory: chapter7, file additionClient.py

The leading content defines the imports and the *main* function. The *main* function creates the *AdditionClient* instance, which generates the requests.

```
"""
Client to the addition service using JSON and JSON Schema.

HTTP client to make requests. Default port is 8303.
"""
try:
    # Python 3
    from urllib.request import urlopen
    from urllib.request import Request
    from urllib.error import HTTPError
except ImportError:
    # Python 2
    from urllib2 import urlopen
    from urllib2 import Request
    from urllib2 import HTTPError
from argparse import ArgumentParser
from json import loads, dumps
from jsonschema import Draft4Validator
import sys

def main ():
    """ Program entry point. """
    AdditionClient ()
```

The *AdditionClient* is declared with the *init* method which sets the *port* (default or from command line parameter), loads the response and error schemas, and ends by calling the method *makeRequests* to generate the *HTTP* requests.

```
25 class AdditionClient:
26     """ Start client for addition service. """
27     def __init__ (self):
28         """ Set port and start client """
29         # process command line for port number
30         self.port = 8303
31         self.processCommand ()
32 
33         # Load JSON Schema to validate result against
34         try:
35             # read the file and convert to a JSON object
36             responseSchema = open ("addResponse_schema.json", "rU").read ()
37             errorSchema = open ("addError_schema.json", "rU").read ()
38         except IOError as e:
39             print ("Error loading schema: " + e.strerror)
40             sys.exit (1)
41 
42         try:
43             self.responseSchema = loads (responseSchema)
44             self.errorSchema = loads (errorSchema)
45         except ValueError as e:
46             print ("Invalid JSON content in schema")
47             sys.exit (1)
48 
49         self.makeRequests ()
```

The *processCommand* method checks the command line arguments for a port option.

```
51     def processCommand (self):
52         """ Get port from command line arguments. """
53         parser = ArgumentParser ()
54         parser.add_argument ("-p", "--port", type=int, dest="port",
55           action="store", help="Port to make requests on")
56         args = parser.parse_args ()
57         if args.port is not None:
58             self.port = args.port
```

The *makeRequests* method constructs the *JSON* content for each of the requests to be made. For each request, it calls *postRequest*.

```
60     def makeRequests (self):
61         """ Make requests with valid and invalid content. """
62         # make a request with valid content
63         data = dumps ({ "number1": 15, "number2": 24 })
64         self.postRequest ("Add 2 numbers", data)
65 
66         # make a request with invalid content
67         data = dumps ({ "number1": 15, "number2": True })
68         self.postRequest ("Add number and boolean", data)
69 
70         # make a request that will get an invalid result
71         data = dumps ({ "number1": 0, "number2": 0 })
72         self.postRequest ("Add two zeros", data)
```

Making the *HTTP* request and processing the *HTTP* response are handled by the *postRequest* method.

- A header for the request is displayed to the console (line 81).
- The request parameters are prepared (lines 82-84).
- The *HTTP* request is made (line 87).

- The *HTTP* response is received, body content read and parsed. (lines 88-90).
- On a successful *HTTP* response, the response is validated against the *addResponse_schema.json* schema (lines 92-93).
  - If the response is validated, the addition result is displayed (line 95).
  - If the response is invalid, the invalid result message is displayed (line 97).
- On an error *HTTP* response, the response is validated against the *addError_schema.json* schema (lines 98-107), with the *additionService* error shown for a validated error response, or a general error if an invalid error response is received.

```
 74     def postRequest (self, name, content):
 75         """
 76         Post a request to the additionService.
 77         Args:
 78             name Request name to display with result
 79             content JSON object to pass to additionService
 80         """
 81         print ("Result for request: " + name)
 82         url = "http://localhost:" + str (self.port) + "/"
 83         dataIn = content.encode ("utf8")
 84         headers = { "Content-type": "application/json" }
 85 
 86         try:
 87             req = Request (url, dataIn, headers)
 88             response = urlopen (req)
 89             dataOut = response.read ().decode ("utf8")
 90             result = loads (dataOut)
 91             try:
 92                 validator = Draft4Validator (self.responseSchema)
 93                 validator.validate (result)
 94 
 95                 print ("  Result = " + str (result["answer"]))
 96             except Exception as e:
 97                 print ("  Invalid result received\n" + str (e))
 98         except HTTPError as e:
 99             data = e.read ().decode ("utf8")
100             result = loads (data)
101             try:
102                 validator = Draft4Validator (self.errorSchema)
103                 validator.validate (result)
104 
105                 print ("  Server error: " + str (result["error"]))
106             except:
107                 print ("Invalid error received")
```

The last two lines handle invocation of the main function when the program is initiated.

```
109 if __name__ == "__main__":
110     main ()
```

After making the requests and displaying their results, the program will end.

Running the Programs

To run the example programs, open two *Terminal* / *Command Prompt* windows. Change directory to the *chapter7* directory under the *bookujs* directory in both windows. The port number option is shown in both commands, but can be omitted from both if the default port is available (8303).

In the first window, use one of the following commands.

```
node additionService.js -p=8303
python additionService.py -p=8303
```

The server will start, and a message displays indicating the port that the server is listening to.

In the second window, use one of the following commands. Note that that port number for the client must match the port number used in the last command.

```
node additionClient.js -p=8303
python additionClient.py -p=8303
```

When the *additionClient* programs runs,

- The *additionService* program will show the requests being received.
- The *additionClient* program will show the results and errors and complete.

The *additionService* will continue running until terminated (with *Ctrl-C*) or the window is closed.

The *additionClient* program can be run multiple times without restarting the *additionService*. The schema content in the program can be changed to see the results from different schema content (valid or invalid).

If you have both *Javascript / Node.js* and *Python* installed, you can run the *additionClient* and *additionService* using different runtime platforms if desired.

Note that using port numbers under 1024 may generate an error if your network configuration limits use of ports between 0 and 1023.

Validation Proxy Server

It can be desirable to validate messages before they are received at the end application or service. A proxy server is a vehicle for inserting an intermediary processor in the message path. Reasons for using a validation proxy server include,

- Security considerations. Identifying and rejecting messages that do not contain acceptable content should be performed as soon as possible, rather than only after the message has reached the application / service instance. Protocol conformance, client identification and similar checks can be augmented with *JSON* content validation.
- Performance and scalability. Distributing the validation processing to a proxy can provide load distribution options. These can be static or dynamic, depending on the environment. Appliances or dedicated accelerators may also be applicable to high volume environments, in addition to general purpose proxy servers.
- Use of specialty proxies that provide *JSON* validation for multiple applications / services. This can include a schema database, optimization of schema processing, and ease of introducing new schemas to a managed schema runtime configuration.

A validation proxy can be implemented in a variety of ways, and can be incorporated as part of a multipurpose proxy server or as a dedicated function.

Proxy Server Example

The following proxy server is a very simple example, meant to express the concept only. Running the example will show the proxy rejecting the request that does not validate, preventing this request from reaching the server. A more robust implementation can add a header between the proxy and server to indicate validation has been performed rather than the server being used as is and repeating the validation step. Note, suitable configuration of the proxy and server networking would also be needed to ensure the header was only accepted from valid proxies.

In the addition service example, the client communicated directly with the service instance, each running in their own process. Rather than create new client and server programs, a proxy server process will be added that is placed between the client and service.

Since the three processes will run on the same computer in the description of running the programs, two port numbers are required (one for the client – proxy communications, and one for the proxy - service communications). The port argument for the client and service will be used, and matched up with the inbound and outbound arguments for the proxy.

The proxy will not modify the messages, thus the client will show the same results whether the proxy is in the message path or not. This is a typical practice for this type of proxy. As such, the proxy implementation will be very similar to the service implementation for message handling, however it won't contain any service logic (in this case, the addition of the numbers).

Javascript/Node.js Implementation

The source code for the proxy *Javascript* / *Node.js* implementation is contained in a single file.

Directory: chapter7, file: additionProxy.js

The proxy implementation borrows from both the service and client programs. The leading content defines inbound and outbound port variables with default values that can be modified by command line arguments.

```
 1 /**
 2  * Proxy for the Addition service, performing validation
 3  * at the proxy..
 4  *
 5  * Starts an HTTP proxy listening for addition requests.
 6  * Inbound default port is 8303, outbound is 8304.
 7  */
 8 var fs = require ("fs");
 9 var http = require ("http");
10 var tv4 = require ("tv4");
11
12 var inboundPort = 8303;
13 var outboundPort = 8304;
14 var requestSchema = null;
15
16 // if module invoked directly, call main
17 if (require.main === module) {
18     main ();
19 }
```

The *main* function loads the request validation schema and sets up the inbound listening port for the proxy.

```
21 /**
22  * Program entry point.
23  */
24 function main () {
25     // process command line for port number
26     processCommand ();
27
28     // Load JSON Schema to validate result against
29     try {
30         var data = fs.readFileSync ("addRequest_schema.json");
31         requestSchema = JSON.parse (data);
32     } catch (e) {
33         console.log ("Error loading request schema: " + e.message);
34         process.exit (1);
```

```
35     }
36
37     // listen for messages on specified port
38     var server = http.createServer (handler);
39     server.listen (inboundPort);
40     console.log ("Addition service proxy");
41     console.log ("  Proxy for port " + outboundPort);
42     console.log ("  Listening on port " + inboundPort);
43 }
```

The *handler* function processes *HTTP* requests received, and for each *JSON* request received, calls the *proxy* function (line 69).

```
45 /**
46  * HTTP request handler
47  * @param response HTTP response object
48  * @param body HTTP body text
49  */
50 function handler (request, response) {
51     // when a message is received, display a message
52     console.log ("Request received");
53
54     // verify the content type is for JSON content
55     var contentType = request.headers["content-type"];
56     if (contentType !== "application/json") {
57         console.log ("Invalid content type: " + contentType);
58     } else {
59         // initialize request content with empty string
60         var body = "";
61
62         // when data is received, add it to request content
63         request.on ("data", function onData (data) {
64             body += data;
65         });
66
67         // when all data is received, process the content
68         request.on ("end", function onEnd () {
69             proxy (response, body);
70         });
71     }
72 }
```

The *proxy* function parses the received content (line 82), validates the content (lines 88-89), and if valid, forwards the request to the service (line 91). If not valid, an error is generated back to the client (lines 93-96).

```
74 /**
75  * Validate and proxy the request.
76  * @param response HTTP response object
77  * @param body HTTP body text
78  */
79 function proxy (response, body) {
80     console.log ("addition body = " + body);
81     // display received content and parse to JSON object
82     var input = JSON.parse (body);
83
84     var result = null;
85     var contentType = { "Content-type": "application/json" };
86
87     // validate against schema
88     var validator = tv4.freshApi ();
89     if (validator.validate (input, requestSchema) === true) {
90         // forward request to additionService
91         forwardRequest (body, response);
```

```
92     } else {
93         result = { "error": "invalid request" };
94         response.writeHead (400, contentType);
95         response.write (JSON.stringify (result));
96         response.end ();
97     }
98 }
```

The *forwardRequest* function creates the request to the service (line 120 and lines 145-146). Handling of the response (lines 121-141) includes verifying the content type (lines 123-124) and accepting the content (lines 126-132). When the response is completely received, it will be forwarded to the client (lines 134-140).

```
100 /**
101  * Forward the request to the additionService.
102  * @param content JSON object to pass to additionService
103  * @param proxyResponse Proxy response object
104  */
105 function forwardRequest (content, proxyResponse) {
106     // create request definition
107     var headers = {
108         "Content-type": "application/json",
109         "Content-length": content.length
110     };
111     var options = {
112         "host": "localhost",
113         "port": outboundPort,
114         "path": "/",
115         "method": "POST",
116         "headers": headers
117     };
118
119     // create request, and accept response messages
120     var request = http.request (options, function (response) {
121         // verify the content type is for JSON content
122         var contentType = response.headers["content-type"];
123         if (contentType !== "application/json") {
124             console.log ("Invalid content type: " + contentType);
125         } else {
126             // initialize response content with empty string
127             var body = "";
128
129             // when data is received, add it to response content
130             response.on ("data", function onData (data) {
131                 body += data;
132             });
133
134             // when all data is received, process the content
135             response.on ("end", function onEnd () {
136                 proxyResponse.writeHead (response.statusCode,
137                     { "Content-type": "application/json" });
138                 proxyResponse.write (body);
139                 proxyResponse.end ();
140             });
141         }
142     });
143
144     // place the content in the body and send the request
145     request.write (content);
146     request.end ();
147 }
```

The last function, *processCommand*, parses the command line arguments and sets the inbound and outbound port variables if found.

```
149 /**
150  * Set ports from command line arguments.
151  */
152 function processCommand () {
153     var command = process.argv.slice (2);
154     command.forEach (function (arg) {
155         if (arg[0] === "-") {
156             var elements = arg.split ("=");
157             var key = elements[0].toUpperCase ();
158             if ((key === "-I") || (key === "--INBOUND")) {
159                 inboundPort = elements[1];
160             } else if ((key === "-I") || (key === "--OUTBOUND")) {
161                 outboundPort = elements[1];
162             }
163         }
164     });
165 }
```

Python Implementation

The source code for the proxy *Python* implementation is contained in a single file.

Directory: chapter7, file: additionProxy.py

The leading content initiates the program, creating an instance of *AdditionProxy*.

```
 1 """
 2 Proxy for the Addition service, performing validation
 3 at the proxy..
 4
 5 Starts an HTTP proxy listening for addition requests.
 6 Inbound default port is 8303, outbound is 8304.
 7 """
 8 try:
 9     # Python 3
10     from http.server import BaseHTTPRequestHandler, HTTPServer
11     from urllib.request import urlopen
12     from urllib.request import Request
13     from urllib.error import HTTPError
14 except ImportError:
15     # Python 2
16     from urllib2 import urlopen
17     from urllib2 import Request
18     from urllib2 import HTTPError
19     from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
20 from argparse import ArgumentParser
21 from json import loads
22 from jsonschema import Draft4Validator
23 import sys
24
25 def main ():
26     """ Program entry point. """
27     AdditionProxy ()
```

The *init* method sets the defaults and calls the method to process the command line arguments. Next, the host tuple is created for the inbound port (line 39). Line 40 creates an instance of *ProxyHTTPServer*, which is a subclass of *HTTPServer*. The new class stores the outbound port, making it accessible to *Handler* instances. Lines 44-48 start the server listener and shutdown the server on a keyboard interrupt (*Ctrl-C*).

```
29 class AdditionProxy:
```

```
30      """ Start server for addition service. """
31      def __init__ (self):
32          """ Set port and start server """
33          # process command line for port number
34          self.inbound = 8303
35          self.outbound = 8304
36          self.processCommand ()
37
38          # listen for messages on specified port
39          host = ("localhost", self.inbound)
40          server = ProxyHTTPServer (host, self.outbound, Handler)
41          print ("Addition service proxy")
42          print ("  Proxy for port " + str (self.outbound))
43          print ("  Listening on port " + str (self.inbound))
44          try:
45              server.serve_forever ()
46          except KeyboardInterrupt:
47              server.shutdown ()
48              server.server_close()
```

The *processCommand* method collects the inbound and outbound port options.

```
50      def processCommand (self):
51          """ Get ports from command line arguments. """
52          parser = ArgumentParser ()
53          parser.add_argument ("-i", "--inbound", type=int, dest="inbound",
54            action="store", help="Inbound port")
55          parser.add_argument ("-o", "--outbound", type=int, dest="outbound",
56            action="store", help="Outbound port")
57          args = parser.parse_args ()
58          if args.inbound is not None:
59              self.inbound = args.inbound
60          if args.outbound is not None:
61              self.outbound = args.outbound
```

The *ProxyHTTPServer* class is defined. The definition includes *HTTPServer* and object to allow the super syntax across *Python* 2 and 3. The *init* method calls the superclass *init* and stores the outbound port, so it can be accessed by *Handler* instances.

```
63 class ProxyHTTPServer (HTTPServer, object):
64      """
65      HTTPServer subclass to hold outbound port
66      """
67      def __init__ (self, host, outbound, handler):
68          super (ProxyHTTPServer, self).__init__ (host, handler)
69          self.outbound = outbound
```

A signification portion of the *Handler* implementation is the same as the *AdditionService*, and will not be repeated here. Lines 113-125 contain the proxy specific content.

- The message validation (lines 113-114) is performed. If the validation does not pass, then the *AdditionService* is not called, instead the error is returned to the client from the proxy.
- The differences are in lines 120-125, where the service logic (addition calculation) is replaced by the request code from the proxy to the service. The response from the service, or errors, are passed through to the client.

```
113                 validator = Draft4Validator (Handler.requestSchema)
114                 validator.validate (dataIn)
115
116                 # make request on outbound port
117                 url = "http://localhost:" + str (self.server.outbound) + "/"
118                 headers = { "Content-type": "application/json" }
119                 try:
```

```
120                     print ("Make request to " + url)
121                     req = Request (url, data.encode ('utf8'), headers)
122                     response = urlopen (req)
123                     dataOut = response.read ().decode ("utf8")
124                     print ("Dataout " + dataOut)
125                     self.send_response (200)
```

Running the Program with the Proxy

For this example, three *Terminal / Command Prompt* windows will be used. Change directory to the *chapter7* directory under the *bookujs* directory in all windows. While the default port number for the client can be used, the server needs to have its port number specified (to match the outbound port on the proxy).

By default, the client and proxy inbound will use port 8303. The proxy outbound uses the default port 8304. The server default is 8303, which must be overridden to match the proxy outbound port.

In the first window, use one of the following commands to start the addition service.

```
node additionService.js -p=8304
python additionService.py -p=8304
```

The server will start, and a message displays indicating the port that the server is listening to.

In the second window, use one of the following commands to start the validation proxy server. Note that the port number for the client must match the inbound port number used in the next command and the outbound port number must match the port from the last command.

```
node additionProxy.js -i=8303 -o=8304
python additionProxy.py -i=8303 -o=8304
```

In the third window, use one of the following commands to start the client. Note that that port number for the client must match the inbound port number used in the last command.

```
node additionClient.js -p=8303
python additionClient.py -p=8303
```

When the *additionClient* program runs,

- The *additionService* program will show the requests being received.
- The *additionProxy* program will show requests being received.
- The *additionClient* program will show the results and errors and complete.

The *additionService* and *additionProxy* will continue running until terminated (with *Ctrl-C*) or the window is closed.

The *additionClient* program can be run multiple times without restarting the *additionService* or *additionProxy*. The *JSON* content in the client program can be changed to see the results from different content (valid or invalid).

If you have both *Javascript / Node.js* and *Python* installed, you can run the *additionClient*, *additionProxy*, and *additionService* using any combination of runtime platforms if desired.

8. Command Line Validation Tool

Throughout *chapter 3*, the *validate* program is used to initiate validation processing of *JSON* content against schemas defined using *JSON Schema*. As a command line tool, is can also be used as part of scripts, enabling the tool to be used for many purposes.

This chapter will cover the source code for both the *Javascript / Node.js* and *Python* versions of the *validate* program. It will also provide some examples for using the program in scripts, and as library functions in other programs.

The *validate* program consists of the following modules,

- *main.js / main.py*. The command line entry point for the program, it processes the command line arguments and initiates the processing.
- *validate.js / validate.py*. Loads the *JSON* content and schema content (from files, database and *HTTP* sources). It initiates the validation processor using the *Tiny Validator* (*Javascript*) or *jsonschema* (*Python*) libraries.

It uses the *safeFile* module for file interactions. The *Javascript / Node.js* version also uses the *format* module. These are covered in *chapter 9*.

The *Javascript/Node.js* and *Python* programs provide the equivalent function, however their implementations differ. The structure of the *Javascript / Node.js* implementation reflects the asynchronous programming model of *Node.js* and the schema loading approach for using T*iny Validator*. The *Python* implementation utilizes a subclassing approach for handling schema loading, and uses the internal *HTTP* schema fetch function from the *jsonschema* library. For readers interested in considering different ways of implementing validation processors, reading through both implementations may be useful.

Entry Point: main.js / main.py

The *Javascript/Node.js* and *Python* implementations have similar implementations.

- Imports are done for dependent modules.
- Programming constructs for main processing are included, supporting the program being called from a command line or script.
- The main program logic processes the command line, verifies the *JSDB* file exists (if being used), invokes the validation process, prints the processing result, and exits with the success / fail exit code.
- A function, *processCommand*, to process the command line. In the *Javascript/Node.js* implementation, this does the processing itself. In the *Python* version, the base library functions from *argparse* are used.

The source code for each follows.

Directory: chapter8/nodejs/jsonvalidate, file: main.js

```
1 /**
2  * Validate JSON file against a JSON Schema
3  *
4  * Usage: jsonvalidate [options] jsonFile schemaFile [referenceFile ...]
```

```
 5  *
 6  * The result will be indicated with the process.exit (n) where,
 7  *   0 indicates successful validation
 8  *   1 indicates validation failed
 9  */
10 var validate = require ("./validate").validate;
11 var fs = require ("fs");
12
13 // if module invoked directly, call the module function
14 if (require.main === module) {
15     main ();
16 }
17
18 /**
19  * Validate JSON per command line arguments.
20  */
21 function main () {
22     // process command line arguments
23     var command = processCommand (process.argv.slice (2));
24
25     if (command.jsdb !== null) {
26         if (fs.exists (command.jsdb) === false) {
27             console.log ("JSDB file specified does not exist");
28             process.exit (1);
29         }
30     }
31
32     // validate content with schema
33     validate (command.json, command.schema, command.ref, command.jsdb,
34         function (code, data, message) {
35         // display message and exit with result code
36         console.log (message);
37         process.exit (code);
38     });
39 }
40
41 /**
42  * Process the command line.
43  * @param {string[]} command Command line arguments.
44  * @returns {object} Object {json, schema, ref, jsdb}.
45  */
46 function processCommand (command) {
47     var showHelp = false; // Flag: help requested?
48     var validCommand = true; // Flag: Valid command?
49
50     // result of command line argument processing
51     var result = {
52         "json" : null,
53         "schema" : null,
54         "ref" : [],
55         "jsdb": null
56     };
57
58     // skip arg0 (program name) and arg1 (script name)
59     command.forEach (function (arg) {
60         // if argument is an option (leads with -)
61         if (arg[0] === "-") {
62             var elements = arg.split ("=");
63             var key = elements[0].toUpperCase ();
64             if ((key === "-J") || (key === "--JSDB")) {
65                 result.jsdb = elements[1];
66             } else if ((key === "-H") || (key === "--HELP")) {
67                 showHelp = true;
68             }
```

```
 69         } else {
 70             // assign positional arguments
 71             if (result.json === null) {
 72                 result.json = arg;
 73             } else if (result.schema === null) {
 74                 result.schema = arg;
 75             } else {
 76                 // accept variable number of ref arguments
 77                 result.ref.push (arg);
 78             }
 79         }
 80     });
 81
 82     // if both files not specified, command is invalid
 83     if ((result.json === null) || (result.schema === null)) {
 84         validCommand = false;
 85     }
 86
 87     // if any errors found or help requested, display usage message
 88     if ((showHelp === true) || (validCommand === false)) {
 89         console.log ("Usage: validate [options] json schema [refs ...]");
 90         console.log ("  json    JSON file to be validated");
 91         console.log ("  schema  JSON Schema file to validate against");
 92         console.log ("  refs    JSON Schema referenced element files");
 93         console.log ();
 94         console.log ("  options:");
 95         console.log ("    -j or --jsdb    JSDB file");
 96         console.log ("    -h or --help    Display this message");
 97
 98         // exit with error code for invalid command or zero if help shown
 99         process.exit ((showHelp === true) ? 0 : 1);
100     }
101
102     return (result);
103 }
104
105 // exports
106 exports.main = main;
```

Of note in the *Javascript/Node.js* implementation is the asynchronous call to *validate* on line 33. The last parameter is the callback function that will be called when the validation processing is complete, which has its implementation on lines 35-37. This asynchronous pattern enables the validation processor to implement schema fetch functions using asynchronous patterns, which is the implementation supported in the *Node.js* runtime libraries.

Directory: chapter8/python/jsonvalidate, file: main.py

```
 1 """
 2 Validate JSON file against a JSON Schema
 3
 4 Usage: jsonvalidate [-options] jsonFile schemaFile [refFiles ...]
 5 Options:
 6   -j    JSDB file containing ref schemas
 7
 8 The result will be indicated with the sys.exit (n) where,
 9   0 indicates successful validation
10   1 indicates validation failed
11 """
12 from argparse import ArgumentParser
13 from os.path import isfile
14 import sys
15 from jsonvalidate.validate import validate
16
```

```
 17 def main ():
 18     """ Validate JSON per command line arguments. """
 19     # process command line arguments
 20     jsonFile, schemaFile, refFiles, jsdbFile = processCommand ()
 21
 22     if jsdbFile is not None:
 23         if not isfile (jsdbFile):
 24             print ("JSDB file specified does not exist")
 25             sys.exit (1)
 26
 27     # validate content with schema
 28     code, data, message = validate (jsonFile, schemaFile, refFiles, jsdbFile)
 29     # display message and exit with result code
 30     print (message)
 31     sys.exit (code)
 32
 33 def processCommand ():
 34     """ Process the command provided. """
 35     # call option processor
 36     parser = ArgumentParser (prog="validate")
 37     parser.add_argument ("jsonFile",
 38       help="JSON file to be validated”)
 39     parser.add_argument ("schemaFile",
 40       help="JSON Schema file to jsonvalidate against")
 41     parser.add_argument ("-j", "--jsdb", dest="jsdbFile", action="store",
 42       help="JSDB file containing ref schemas")
 43     parser.add_argument ("refFiles", nargs="*",
 44       help="JSON Schema files with referenced elements")
 45     args = parser.parse_args ()
 46
 47     # return command line parse results
 48     if "jsdbFile" not in args:
 49         args["jsdbFile"] = None
 50     return args.jsonFile, args.schemaFile, args.refFiles, args.jsdbFile
 51
 52 if __name__ == "__main__":
 53     main ()
```

The *Python* implementation of the *validate* call (line 28) uses a synchronous call and a multiple return value syntax for the results. The use of a synchronous call is consistent with the implementation of the *jsonschema* and *Python* libraries used.

Validation Processing: validate.js / validate.py

The *Javascript/Node.js* and *Python* implementations perform the same function, but their implementations are very different. This shows the flexibility available to implementers choosing how to best fit each runtime platform.

Javascript / Node.js Version

The *Javascript/Node.js* validation process has three distinct steps – receiving the instructions, assembling and organizing the content, and running the validation. In the second of these steps, it retrieves the data and schemas, determines whether all the required content is present, and invokes the validation processor.

The *Tiny Validator* library provides a flexible API for interacting with the schema processor, including interactions at intermediate data stages. This allows the validation processor to be used against partially complete schema models to determine what is missing, which can subsequently be populated before running the validation.

Directory: chapter8/nodejs/jsonvalidate, file: validate.js

The leading content defines the error constants and messages and the module variable (*jsdbData*) to hold the *JSDB* content for referenced schema content.

```
 1 /**
 2  * JSON Schema validation preparation.
 3  *
 4  * Load data and schema content
 5  * - resolve local file references
 6  * - resolve database references (jsdb)
 7  * - resolve remote HTTP references
 8  */
 9 var http = require ("http");
10 var tv4 = require ("tv4");
11 var safeFile = require ("ujs-safefile").safeFile;
12 var format = require ("ujs-format").format;
13
14 // message numbers and formats
15 var VALID = 0;
16 var INVALID_JSON = 200;
17 var MISSING_ID = 201;
18 var FETCH_ERROR = 300;
19 var VALIDATION_ERROR = 301;
20
21 var MSG_READ_ERROR = "Error reading {0}: {1}";
22 var MSG_VALID_JSON = "JSON content in file {0} is valid";
23 var MSG_INVALID_JSON = "Invalid JSON in file: {0}. Error: {1}";
24 var MSG_MISSING_ID = "Missing Id in Reference Schema {0}";
25 var MSG_FETCH_ERROR = "Error fetching {0}: {1}”;
26
27 // JSDB data
28 var jsdbData = null;
```

The *validate* function will be shown in sections for clarity. The first section includes the comment, function declaration, and variable initialization.

```
30 /**
31  * Process inputs for validation, including fetching external schema
32  * content. If all inputs are valid, call validation processor
33  * (in runValidate).
34  * @param {string} dataFile File with JSON content to validate.
35  * @param {string} schemaFile File containing JSON Schema.
36  * @param {string[]} refFiles Array of files for schemas referenced.
37  * @param {string} jsdbFile File containing JSDB content.
38  * @callback {callback} callback with object containing
39  *     {integer} code: VALID or error constant.
40  *     {string} data: data read for VALID result.
41  *     {string} message: result text.
42  */
43 function validate (dataFile, schemaFile, refFiles, jsdbFile, callback) {
44     var data = null;
45     var schema = null;
46     var refs = [];
47     var result = null;
48     var message = null;
```

Load the data and schema content.

```
50     // read data file
51     result = readJsonFile (dataFile);
52     if (result.code !== VALID) {
53         message = format (MSG_READ_ERROR, dataFile, result.message);
54         return callback (result.code, null, message);
55     }
```

```
56      data = result.data;
57
58      // read schema file
59      result = readJsonFile (schemaFile);
60      if (result.code !== VALID) {
61          message = format (MSG_READ_ERROR, schemaFile, result.message);
62          return callback (result.code, null, message);
63      }
64      schema = result.data;
```

If reference schemas are specified, the array of reference schemas will be read. Each is verified to include a top level *id* property, otherwise the validator will not be able to associate it with a *$ref URI*.

```
66      // read set of reference files
67      if (refFiles !== null) {
68          for (var ctr1 = 0; ctr1 < refFiles.length; ctr1 ++) {
69              var file = refFiles[ctr1];
70              result = readJsonFile (file);
71              if (result.code !== VALID) {
72                  message = format (MSG_READ_ERROR, file, result.message);
73                  return callback (result.code, null, message);
74              }
75
76              var ref = result.data;
77              if (ref.id === undefined) {
78                  message = format (MSG_MISSING_ID, file);
79                  return callback (MISSING_ID, null, message);
80              }
81              refs.push ({ "uri":ref.id, "schema":ref, "error":null });
82          }
83      }
```

If a *JSDB* source is being used, it will be read into the *jsdbData* variable.

```
85      // read JSDB file
86      if (jsdbFile !== null) {
87          result = readJsonFile (jsdbFile);
88          if (result.code !== VALID) {
89              message = format (MSG_READ_ERROR, jsdbFile, result.message);
90              return callback (result.code, null, message);
91          }
92          jsdbData = result.data;
93      }
```

The next section creates a fresh validation processor, and populates the schema content (top level schema and all reference schemas).

```
 95      // reset validator and add loaded schemas
 96      var validator = tv4.freshApi ();
 97      validator.addSchema ("", schema);
 98      for (var ctr2 = 0; ctr2 < refs.length; ctr2 ++) {
 99          validator.addSchema (refs[ctr2].uri, refs[ctr2].schema);
100      }
```

The last section allow reference schema content to be loaded from other *URI* sources, *http* and *jsdb*, as specified in *$ref URIs*.

The *fetchSchemaContent* function can be called recursively. Each recursion into *fetchSchemaContent* will retrieve an additional layer of schema content. When all schema content has been received, the callback function provided to *fetchSchemaContent* will be called. Line 104 is the initial call to *fetchSchemaContent*, and its callback will be invoked when all layers of schema content have been resolved.

If the fetch processing was successful, the validation step will be invoked (lines 105-106). If any errors occurred, the error message will be generated and returned through the callback (lines 107-117).

```
102     // resolve refs from other locations (http, jsdb). Call validation
103     // when schema content is complete.
104     fetchSchemaContent (validator, refs, function (successful) {
105         if (successful) {
106             runValidate (dataFile, data, schema, refs, callback);
107         } else {
108             // return URIs in error
109             var m3 = "";
110             for (var ctr3 = 0; ctr3 < refs.length; ctr3 ++) {
111                 var ref = refs[ctr3];
112                 if (ref.error !== null) {
113                     m3 += format (MSG_FETCH_ERROR, ref.uri, ref.error) + "\n";
114                 }
115             }
116             return callback (FETCH_ERROR, null, m3);
117         }
118     });
119 }
```

The *readJsonFile* function reads the content of the file specified and returns its data content after verifying that the content is valid *JSON*.

```
121 /**
122  * Read file and verify it contains JSON content.
123  * @param {string} file Path/name of file to read.
124  * @returns {object} Result { code, data, message }
125  */
126 function readJsonFile (file) {
127     var data = null;
128     var code = null;
129     var message = null;
130
131     try {
132         var content = safeFile.readFileSync (file);
133         try {
134             data = JSON.parse (content);
135             code = VALID;
136         } catch (e1) {
137             code = INVALID_JSON;
138             message = format (MSG_INVALID_JSON, file, e1.message);
139         }
140     } catch (e2) {
141         code = e2.code;
142         message = e2.message;
143     }
144
145     return ({code:code, data:data, message:message});
146 }
```

The *fetchSchemaContent* function is a recursive function that loads schema content one layer at a time. As each layer is loaded, the function checks to see if any new references were introduced, and if so, it processes through the new layer. Once there are no outstanding references, the function completes and sends back its results to the calling function through a callback. Given the length of the function, the source code will be shown in sections.

The *fetchSchemaContent* function determines whether any *URIs* are not yet resolved, and if so, select the retrieval mechanism to load the content. Processing of *http* / *https* and *jsdb URIs* is supported. First is the comment and function declaration.

The first step, getting the list of missing *URIs*, is preformed by calling the *getMissingUris* function in the *Tiny Validator*, and if none are missing, calling the completion callback. Since no content is missing, the callback indicates success with the argument *true*.

```
148 /**
149  * Fetch schema content for the current depth. Can be called recursively
150  * to create the fully populated schema.
151  * @param validator Validation processor instance
152  * @param refs Array containing full set of referenced schemas.
153  * @param callback Carries result of schema processing (true, false).
154  */
155 function fetchSchemaContent (validator, refs, callback) {
156     // if no missing URIs then end fetch
157     var missingUris = validator.getMissingUris ();
158     if ((missingUris === null) || (missingUris.length === 0)) {
159         return callback (true);
160     }
```

The next section assembles the list of outstanding references by *URI* type (*jsdb:* or *http:/https:*).

```
162     // assemble list of missing URIs by resource type
163     var jsdbList = [];
164     var httpList = [];
165     for (var ctr1 = 0; ctr1 < missingUris.length; ctr1 ++) {
166         // get protocol from URI (portion up to the first colon)
167         var uri = missingUris[ctr1];
168         var protocol = uri.substring (0, uri.indexOf (":"));
169 
170         // assign to appropriate resource list
171         if (protocol === "jsdb") {
172             jsdbList.push (uri);
173         } else if ((protocol === "http") || (protocol === "https")) {
174             httpList.push (uri);
175         }
176     }
```

Fetching *JSDB* resource content uses the synchronous *jsdbFetch* function. After each fetch, the fetch result is added to the *refs* list, and on successful fetches the schema is added to the schema processor. After all fetches, if any errors have occurred (lines 189-192), then the callback is called with a *false* result.

```
178     // fetch schemas from JSDB resource, on any error end fetch
179     var jsdbOkay = true;
180     for (var ctr2 = 0; ctr2 < jsdbList.length; ctr2 ++) {
181         var ref = jsdbFetch (jsdbList[ctr2]);
182         refs.push (ref);
183         if (ref.schema !== null) {
184             validator.addSchema (ref.uri, ref.schema);
185         } else {
186             jsdbOkay = false;
187         }
188     }
189     // if any errors, end processing
190     if (jsdbOkay === false) {
191         return callback (false);
192     }
```

Fetching *HTTP* content (lines 194-238). If there are no *HTTP URIs* for this layer, then the schema fetching is complete for this layer, and *fetchSchemaContent* is recursively called to initiate processing of the next layer. The callback function passed will propagate the result received to the calling function.

```
194     // If no HTTP requests, continue with next depth processing
195     if (httpList.length === 0) {
196         fetchSchemaContent (validator, refs, function (result) {
```

```
197             // propagate result back through caller chain
198             return callback (result);
199         });
```

If there are *HTTP* fetches to be resolved, *HTTP URI* fetch processing (lines 200-238) uses an asynchronous technique, allowing the *HTTP* requests to be executed in parallel.

Lines 202-203 initialize the *results*, and *newRefs* variables. The *results* variable is used to determine when all fetches are complete by comparing the number of responses received to the number of requests made. The *newRefs* variable holds a list of result objects, collecting the status from each request made.

Lines 204-232 contain the *handler* callback. The *handler* is the callback function that will be invoked by the *httpFetch* function when the *HTTP* response is received. For each response, line 205 adds the content/error to the list of response results (*newRefs*). Next, it checks to see whether all requests have been completed (*results* counter versus requests list length).

When all requests have responses, all successful requests will have their schemas added to the *refs* list and the schema processor (lines 209-220).

Lines 222-230 complete the callback function. If all requests were completed successfully, the *fetchSchemaContent* function will be called recursively to initiate processing of the next layer (lines 223-227). When called, the callback function passed will propagate the result received to the calling function. However, if any errors occurred (lines 228-230), the callback will be called with the *false* value.

The last section of the *fetchSchemaContent* function (lines 234-237) contain the calls to the *httpFetch* function for each *HTTP URI* to retrieve a schema from. The *handler* function defined above is passed as the callback to be called by the *httpFetch* function after receiving each response.

```
200     } else {
201         // process HTTP fetches (async)
202         var results = 0;
203         var newRefs = [];
204         var handler = function (uri, schema, error) {
205             newRefs.push ({ "uri":uri, "schema":schema, "error":error });
206 
207             // when all HTTP requests complete, process all collected
208             results ++;
209             if (results === httpList.length) {
210                 // add all success/fail records to refs
211                 var httpOkay = true;
212                 for (var ctr3 = 0; ctr3 < newRefs.length; ctr3 ++) {
213                     if (newRefs[ctr3].schema !== null) {
214                         var newRef = newRefs[ctr3];
215                         refs.push (newRef);
216                         validator.addSchema (newRef.uri, newRef.schema);
217                     } else {
218                         httpOkay = false;
219                     }
220                 }
221 
222                 // if no errors, process next depth, else end
223                 if (httpOkay) {
224                     fetchSchemaContent (validator, refs, function (result) {
225                         // propagate result back through caller chain
226                         return callback (result);
227                     });
228                 } else {
229                     return callback (false);
```

```
230                 }
231             }
232         };
233
234         // initiate processing for the set of fetch requests (async)
235         for (var ctr4 = 0; ctr4 < httpList.length; ctr4 ++) {
236             httpFetch (httpList[ctr4], handler);
237         }
238     }
239 }
```

The *runValidate* function creates a fresh validation processor, populates it with the *$ref* schema content collected (lines 255-257) and initiates the validation processor with the data to be validated (line 260). Lines 262-272 contain the message construction and callbacks for successful and failure results.

```
241 /**
242  * Call the validation processor
243  * @param {string} file Name of file being validated.
244  * @param {object} data JSON to validate.
245  * @param {object} schema JSON Schema to validate against.
246  * @param {object[]) refs JSON referenced schemas.
247  * @param {callback} callback with object containing
248  *      {integer} code: VALID or error constant.
249  *      {string} data: data read for VALID result.
250  *      {string} message: result text.
251  */
252 function runValidate (file, data, schema, refs, callback) {
253     // start with fresh instance and add referenced schemas
254     var validator = tv4.freshApi ();
255     for (var ctr2 = 0; ctr2 < refs.length; ctr2 ++) {
256         validator.addSchema (refs[ctr2].uri, refs[ctr2].schema);
257     }
258
259     // validate data against schema specified
260     var result = validator.validate (data, schema);
261
262     // if validation successful, return data and valid message
263     if (result === true) {
264         var m2 = format (MSG_VALID_JSON, file);
265         callback (VALID, data, m2);
266     } else {
267         // if validation failed, display error information
268         var message = "Invalid: " + validator.error.message;
269         message += "\nJSON Schema element: " + validator.error.schemaPath;
270         message += "\nJSON Content path: " + validator.error.dataPath;
271         callback (VALIDATION_ERROR, null, message);
272     }
273 }
```

The *jsdbFetch* function searches the loaded *JSDB* schemas to find a matching *URI*. If found the schema is returned. If not found, an error is returned.

```
275 /**
276  * Fetch schema content from schema database (URI jsdb:).
277  * @param {string} uri URI to resolve.
278  * @returns {object} Schema object (uri, schema, error).
279  */
280 function jsdbFetch (uri) {
281     if (jsdbData !== null) {
282         // if URI found in database, return schema for the URI
283         for (var ctr = 0; ctr < jsdbData.length; ctr ++) {
284             if (jsdbData[ctr].id === uri) {
285                 return ({ "uri":uri, "schema":jsdbData[ctr], "error":null });
286             }
```

```
287         }
288     }
289
290     // no match found, return error result
291     var message = "No schema found for URI: " + uri;
292     return ({ "uri":uri, "schema":null, "error":message });
293 }
```

The *httpFetch* function processes an asynchronous request / response interaction. After processing a successful response (lines 303-311), the callback is called with the schema content. If an error occurs, the callback is called with an error (lines 315-317).

```
295 /**
296  * Fetch the content from a URI using HTTP
297  * @param {string} uri URI of the content.
298  * @param {callback} callback (uri, data, null) or (uri, null, error).
299  */
300 function httpFetch (uri, callback) {
301     // make request to server, with callback function to collect response
302     var request = http.get (uri, function (response) {
303         var data = "";
304         // collect data received from server
305         response.on ("data", function onData (d) {
306             data += d;
307         });
308         // when all data received, send schema to callback
309         response.on ("end", function onEnd () {
310             callback (uri, data, null);
311         });
312     });
313
314     // on error, send error to callback
315     request.on ("error", function (e) {
316         callback (uri, null, e);
317     });
318 }
```

The last section contains the exports for the error codes.

```
320 // exports
321 exports.VALID = VALID;
322 exports.INVALID_JSON = INVALID_JSON;
323 exports.MISSING_ID = MISSING_ID;
324 exports.FETCH_ERROR = FETCH_ERROR;
325 exports.VALIDATION_ERROR = VALIDATION_ERROR;
326 exports.validate = validate;
```

With many possible reference schema to retrieve, and the mix of synchronous and asynchronous techniques shown, the utility of using an array of objects that contains key, data, and error properties to collect results is shown. This mechanism permits retrieved data and errors to be managed in a uniform manner without requiring complex flow control interactions with the caller or across different fetch activities.

Python Version

Extending the *Python* validation processing of *jsonschema* is done by subclassing part of the validation processor, incorporating additional function into the execution path of the validation processor itself. In contrast, the *Javascript* / *Node.js* approach extended the function of the validation processor by populating the data model of the validation processor using the *Tiny Validator APIs* and then invoking the validation processor.

Directory: chapter8/python/jsonvalidate, file: validate.py

The leading section defines the message constants.

```
"""
JSON Schema validation preparation.
 - Load data and schema content
 - resolve local file references
 - resolve database references (jsdb)
"""
from safefile import readFile, SafeFileError
from jsonschema import Draft4Validator, RefResolver
import json

# message numbers and formats
VALID = 0
INVALID_JSON = 200
MISSING_ID = 201
FETCH_ERROR = 300
VALIDATION_ERROR = 301

MSG_READ_ERROR = "Error reading {0}: {1}"
MSG_INVALID_JSON = "Invalid JSON in file: {0}. Error: {1}"
MSG_MISSING_ID = "Missing Id in Reference Schema {0}"
MSG_FETCH_ERROR = "Error fetching {0}: {1}"
MSG_VALID_JSON = "JSON content in file {0} is valid"
```

The next section is the *JsdbResolver* class definition, which extends the *RefResolver* class from the *jsonschema* library. This class will be called when schema content needs to be loaded by the schema processor. The init method calls the superclass constructor and stores the *JSDB* content in the class instance.

```
class JsdbResolver (RefResolver):
    """
    Extends jsonschema resolver with the following:
    - addSchema to add statically defined schemas
    - support for jsdb: URI for database schemas
    """
    def __init__ (self, baseURI, referer, jsdb):
        """ Initialize jsdb and call superclass init """
        super (JsdbResolver, self).__init__ (baseURI, referer)

        # Store JSDB content in memory
        self.jsdb = jsdb
```

The *addSchema* method adds a schema to the set of stored schemas.

```
    def add_schema (self, uri, schema):
        """ Add a schema to the stored list of schemas """
        self.store[uri] = schema
```

The *resolve_jsdb* method searches the stored schemas for one with a matching *URI*. Returns the schema found, or None if no match found.

```
    def resolve_jsdb (self, uri):
        """ Fetch a schema from the JSDB database. """
        result = None
        # if database available
        if self.jsdb is not None:
            # find schema matching id in database and add schema
            for schema in self.jsdb:
                if schema["id"] == uri:
                    result = schema
                    break
        return result
```

The *resolve_remote* method is called when the schema processor needs to retrieve a schema using a *URI*. By overriding this method, the resolver for the *jsdb: URI* is added. The custom resolve logic for *jsdb: URIs* is executed here. For other *URIs,* the superclass *resolve_remote* method is called.

```
53      def resolve_remote (self, uri):
54          """
55          Overrides superclass resolve_remote, processing "jsdb:" URI,
56          otherwise calls superclass to fetch the schema.
57          """
58          if uri[0:5] == "jsdb:":
59              document = self.resolve_jsdb (uri)
60
61              # duplicate caching logic from superclass
62              if self.cache_remote:
63                  self.store[uri] = document
64              return document
65          else:
66              return RefResolver.resolve_remote (self, uri)
```

Next is the *validate* function which is described in sections. The first is the declaration and function comment.

```
68 def validate (dataFile, schemaFile, refFiles, jsdbFile):
69      """
70      Perform validation of JSON content with the JSON Schema.
71
72      Args:
73        dataFile (str): File with JSON content to validate.
74        schemaFilename (str): File containing JSON Schema.
75        refFiles (list of str): List of files for schemas referenced.
76        jsdbFile (str): File containing JSDB schemas referenced.
77      Returns:
78        code (int): VALID or error constant.
79        data (str): data read for VALID result.
80        message (str): message text.
81      """
```

The second section reads the data and schema files.

```
82      # read data file, returning error if not valid
83      code, data, message = _readJsonFile (dataFile)
84      if code != VALID:
85          return code, None, MSG_READ_ERROR.format (jsdbFile, message)
86
87      # read schema file, returning error if not valid
88      code, schema, message = _readJsonFile (schemaFile)
89      if code != VALID:
90          return code, None, MSG_READ_ERROR.format (jsdbFile, message)
```

The third section loads the *JSDB* content (if specified) and then creates the resolver using the *JsdbResolver* class definition.

```
 92      # load JSDB file, or set to empty if not specified
 93      if jsdbFile is None:
 94          jsdb = {}
 95      else:
 96          code, jsdb, message = _readJsonFile (jsdbFile)
 97          if code != VALID:
 98              return code, None, MSG_READ_ERROR.format (jsdbFile, message)
 99
100      # create custom resolver
101      resolver = JsdbResolver ("", schema, jsdb)
```

The fourth section reads the reference files. The schemas from the reference files are added to the custom resolver.

```
    # read reference schema files, returning error if any not valid
    if refFiles is not None:
        for refFile in refFiles:
            code, ref, message = _readJsonFile (refFile)
            if code != VALID:
                return code, None, MSG_READ_ERROR.format (jsdbFile, message)
            if "id" not in ref:
                return MISSING_ID, None, MSG_MISSING_ID.format (refFile)
            resolver.add_schema (ref["id"], ref)
```

The last section creates the validation processor instance with the custom resolver, and then calls it to process the *JSON* data. The success or failure message will be returned.

```
    # run validation, returning data if successful
    try:
        # create validator with custom resolver, call it
        validator = Draft4Validator (schema, resolver=resolver)
        validator.validate (data)
        return VALID, data, MSG_VALID_JSON.format (dataFile)
    except Exception as e:
        # if validation failed, return error information
        return VALIDATION_ERROR, None, e
```

The schema processor will interact with the schema content managed by this program logic through the custom resolver when it calls the *resolve_remote* method.

The last method, *readJsonFile,* reads the content of the specified file and verifies that the content is valid *JSON*.

```
def _readJsonFile (file):
    """
    Read file and verify it contains JSON content
    Args:
        file (str): File to read
    Returns:
      code (int): VALID or error constant.
      data (str): data read for VALID result.
      message (str): message text.
    """
    try:
        data = readFile (file)
        try:
            jsonData = json.loads (data)
            return VALID, jsonData, None
        except ValueError as e:
            return INVALID_JSON, None, MSG_INVALID_JSON.format (file, e)
    except SafeFileError as e:
        return e.code, None, e.message
```

Using the Tools in Shells and Scripts

Command line tools can be used for interactive shell commands or in scripts. These uses can improve productivity and support automation of processes from build to quality assurance testing to field support.

A good example of a useful shell command use is a syntax scan of a set of files. Open a *Terminal* / *Command Prompt* and switch to the *chapter2* directory.

On *Linux* using the *bash* shell, use the command.

```
for file in *.json ; do jsonsyntax $file ; done
```

On *Windows,* use the command.

```
for %f in (*.json) do jsonsyntax %f
```

The result of running the command will be a list of executions of the *jsonsyntax* tool against each of the files matching the filter **.json* with the results for each. This provides a quick way to verify that all the files contain valid *JSON* content.

Beyond the command line interactions, scripting can work with results from program executions to provide automation. For example, printing a successful message if all files are valid *JSON*, or showing errors only for those that contain invalid *JSON* content.

The result of the *jsonsyntax* and validate execution are returned as the exit code for the program. 0 (zero) is returned for a successful execution, and 1 (one) is returned for an unsuccessful execution. This allows *jsonsyntax* and validate to be used in a conditional expression within a script. For example, in a *Linux* shell script.

Directory: chapter8, file: syntaxCheck.sh

```
#!/bin/sh
valid="TRUE"
for f in ../chapter2/*.json
do
  ./syntax.sh $f >nul
  if [ $? -eq 1 ]
  then
    echo File: $f invalid
    valid="FALSE"
  else
    echo File: $f valid
  fi
done

if [ $valid -eq "TRUE" ]
then
  echo All files valid.
  exit 0
else
  exit 1
fi
```

The equivalent in a *Windows* batch file follows.

Directory: chapter8, file: syntaxCheck.cmd

```
@echo off
set valid="TRUE"
for %%f in (..\chapter2\*.json) do (
  call syntax %%f >nul
  if errorlevel 1 (
    echo File: %%f invalid
    set valid="FALSE"
  ) else (
    echo File: %%f valid
  )
)
if %valid%=="TRUE" (
  echo All files valid.
)
```

In both cases, the following steps occur in the script processing.

- A variable (*valid*) is set to track whether all files checked pass the syntax check.
- A for loop executes once for each file with the extension *json* in the *chapter2* directory.
  - If the file contains valid *JSON* content, the valid message is displayed.

- If the file does not contain valid *JSON* content, the invalid message is displayed and the overall flag is set to *false*.
- After checking all files, a final message is displayed if all files checked were valid.

Similar scripts can be used in build processes. As content and schema files are changed,

- Each can be verified to have correct *JSON* syntax
- Each content file can be validated against its schema to ensure no content errors have been introduced.

Scripts can also be used to perform validation on any content that is introduced to the environment. This can be useful for validating content received from external sources, or content generated by programs.

9. Designing Software to Use JSON Files

There are three design areas for the use of *JSON* file use in software programs.

- Validating content to be used by the software program.
- Managing the representation of the data in its persistent and in-memory states.
- Handling error conditions, including errors related to persistence.

The first item applies to all uses of *JSON* (message exchange or persistent storage). The second and third items relate to persistent storage considerations.

Validation in Programs

When *JSON* content is read, the content should be validated before proceeding with its use in the program. The validation can be performed at different levels, suitable for the program needs.

- The content may be validated only for syntax correctness. For programs dealing with generic *JSON* content, or using serialization/deserialization with no external access to the content, this may be sufficient. Using the built in *JSON* parser available with many language runtime libraries will suffice, as shown in the syntax checking examples in *chapter 2*.
- Validation of the structure of the *JSON* content, but not its content. This ensures that the content received meets the expected structure, but leaves interpretation of the content to the program. Programs that accept arbitrary content, such as a web page or content types that have variable data elements, can ensure that the required content is correct, and arbitrary content is correctly delineated. This can include the use of *JSON Schema* for the portions of the content that are required.
- Full content validation of the *JSON* content, providing the validation processor with a well defined schema to use.

Even in the last case, there may be some validation of the content that is program specific or rely on other content. Thus *JSON Schema* may be augmented with program logic to complete the full validation for some content. For example, the *JSON* content may contain an employee serial number as a property within an object, and its content may be constrained to be a five digit integer. The *JSON Schema* validation can verify that a presented employee serial number of 12345 is a valid representation, but the program may further validate the value by doing a query to an employee database to verify that 12345 is an active employee.

In Memory State of JSON Content

Javascript and *Python* examples are shown throughout the book.

- Using *Javascript*, as may be expected, the in-memory representation of the *JSON* content and its manipulation are consistent with *Javascript* objects, arrays, and data types. Serialization and deserialization is very uniform between the representations.
- Using *Python*, the representation is very recognizable, using the dictionary representation for the content. The interfaces for serialization and deserialization are also easy to use.

Other programming languages have similar conversions as seen with *Python*. Data types and structures to handle object/array elements will vary, but typically the mapping is readily recognizable. If the data represented in the *JSON* content is interacted with throughout a program (rather than just load at start and save at end), then using a class (or equivalent) to provide an interface to access / manipulate the data is often useful.

- Load and save methods/functions always have access to the current state.
- Data changes can ensure consistency of the state where applicable (e.g., changing an identifier in one element can be cascaded to its affiliated elements).

The design patterns for many programs will be similar to interactions with a database, where the processes that interact with the data will perform their interactions through a model object or function library. In these cases, the *JSON* content and its persistent storage are participants in typical database patterns (whether a database, file, or other resource, is ultimately the persistent store).

Validation of in-memory representations of *JSON* content can be performed at any time. For programs that support import of data from other sources, or user entry of content, validation of content as part of updating the in-memory representation is often useful for recognizing and resolving errors at the point they are introduced. For programs that support *Undo* functions, this can be helpful in always providing a valid representation for each *Undo* checkpoint.

Persistent State Choices

When considering general data management, including data files and databases, the *JSON* file and *JSON Schema* discussions have some independent considerations.

- *JSON Schema* can apply to *JSON* content stored in any medium. It can be a native capability of the persistence function, or the program can apply the schema to content sent to or received from the persistence function.
- The location of the schema logic is not fixed. If *JSON* content moves from file storage (with program logic applying validation) to a database that supports schema validation as a native feature, the schema logic location moves, but the schema definition remains the same.

In the *JSON* files examples to this point, the discussion has focused on configuration files. However, especially with design topics like auto-save, the applicability of these capabilities to general data files can be envisioned.

However, just like using a full blown database is overkill for most configuration file scenarios, *JSON* files have places where they are suitable for use and where they are not. The following criteria tend to exclude the use of files as the persistent storage choice.

- Programs that require physical media storage updates on every data change. These programs also are likely to be changing a small portion of the content with each change, requiring efficient write operations to individual file segments.
- Programs with large data sets that use indexes to efficiently find and store data. Data storage libraries and databases provide the additional function required for index and search.
- Data sizes that exceed memory available will require options that do not rely on basic in-memory representations of the data.

These considerations leave a lot of space for data management for smaller data sets that do not have transactional requirements. Some criteria that bring *JSON* files into consideration are,

- Human readable, and editable, using a text editor (this is usually a positive, but can be a negative if you prefer not having easy edit access outside the program).
- Name-value pair structure is durable, extensible, and not prone to positional errors of data formats like comma separated value.
- Relatively easy to translate records (into arrays) and fields (into object properties) from current data sources.
- Validation using *JSON* Schema, providing the benefits of structured data without requiring use of a particular persistent storage choice.

Many of these criteria are shared with the use of *XML*. Whether *JSON* or *XML* is applicable will likely incorporate criteria from the domain the program is to be used in. For those domains that have existing schema and document content that already use one or the other, the value of switching is less than for domains that are adopting new.

Program Interaction Models for Persistent Storage

Each program has its own requirements for how, and when, persistent storage is used. These requirements will inform the implementation of the persistent storage function.

Read at Start, Write Before Exit

Fairly common in utility programs and programs that self manage their own data resources. On program load, the *JSON* content is read into memory. During execution, the in-memory representation may be updated. When the program has completed processing related to the data, the persistent state is updated. If no relevant changes to the data were made, the save step can be skipped.

User / Program Initiated Read and Write

The previous example had one well defined read point and one well defined write point, at the initiation and termination of the program respectively. This also provides a predictable context for recovery procedures, since it has a repeatable relationship between the data file state and the program execution. This example separates the read / write operations from the program lifecycle, allowing thee operations to be initiated by events (user or program initiated), and possibly more than once during the program execution.

In the program design, the context of a backup file is no longer assumed to be associated strictly with the program execution itself, since more than one version of the file can be produced by a single execution. Recovery will therefore be to the last save event, which may or may not be the same as the last program execution.

As a variation, the program could choose to load a default file as part of its startup processing. However, this would be considered a program initiated read event in this example, rather than the explicit function described in the previous example.

When the write operations are initiated, the program can determine the suitability of the current state of the data for writing. If the state is not suitable, an appropriate action can be initiated by the

program to ensure the persistent state written will always be acceptable. This step can include validation of the content against a schema, data consistency checks, and other logic. With this verification capability available, the writes can be to the permanent persistent state resources.

Periodic Auto-Save

For programs that run a long time, or where many changes occur between updates to the permanent persistent resource, creating interim checkpoints can provide a useful function to speed recovery in the case of failure.

Auto-save provides a snapshot capability that allows data changes during the course of a program execution to be captured, so that in the event of failure the data changes are not lost up to the point of the last auto-save. However, since these changes have not been completed to the point where the user or program is ready to initiate a write, the changes cannot be made to the permanent persistent resource.

Also, given the periodic nature of the save activity (time, actions, or other metric), the snapshot data may or may not be in a consistent state when it is written. For example, in the address examples (chapters 3 and 4), a new address may be created that specifies a country that has not yet been defined in the common postal information data. The auto-save may save this interim state, so the address is not lost, but until the country is defined, the auto-save data is not in a consistent state.

One of the design challenges for auto-save is how to implement the recovery process, and this in turn will provide insight into what programs auto-save is suitable for, and which it is not. Some considerations,

- Auto-save can be triggered on a timer, a number of changes, or when a particular action is taken (e.g.,the program presents a preferences dialog and the user completes their interaction with the dialog).
- Auto-save typically writes its content to an auto-save specific file, not the permanent file, which must be accounted for in the recovery design. While these are considered temporary resources, these files usually have similar robustness requirements for recovery purposes that permanent resources do.
- Auto-save is often transparent to the end user, so it may not be clear at what point the auto-save content relates to the point that the software failed. This issue will vary in importance and obviousness from program to program, but should be considered in the recovery process, especially those that involve user choices.

From these considerations, the use of auto-save has some very distinct differences from the first example. Auto-save is often more suited to interactive programs, where the recovery process can include user choices and interactions.

Persistent Storage of JSON Content

The persistent storage options for *JSON* content include standalone resources, such as a file in a file system, or in managed resources, such as an entry in a database. How the persistent data is interacted with varies from program to program.

Simple File Storage

The simple scenario: read the *JSON* content when the program starts. Program logic or user interaction with the program allows changes to the in-memory representation of the content. Write the updated content to the file during one of the last processing steps of the program.

This can be suitable for programs that can automatically recreate their data, or for which the data is trivial. Any time the data is changed during the running of the program (or even every time if change tracking is not part of the program), the file will be overwritten.

The next step up, when some basic recovery protection is desired, is to keep a backup of the original file rather than overwriting it. This addresses two weaknesses in the original approach – loss of data should a failure occur during the write process, and saving unwanted changes that are realized only after the program has ended and the changes already saved. The recovery process in this scenario is manual – the user / administrator can delete the new file and rename the backup file to the original name. The process for this approach would be,

- Delete current backup if it exists (e.g.,*employee.json.bak*)
- Rename the original to the backup name (e.g.,*employee.json > employee.json.bak*)
- Save the new content to the original name (e.g.,*employee.json*)

This basic scenario can be extended to similar scenarios, such as allowing the user to direct the save action rather than just saving at the end of processing.

For auto-save scenarios however, the approach of replacing the original file or backup file is insufficient. Since the auto-save activity may happen when the data is inconsistent, and/or the user may not be able to determine what changes were still outstanding should the program be restarted with the auto-saved data. For an auto-save scenario, the auto-save version of the data should be stored in an independent file (e.g.,*employee.json.autosave*), which should be cleaned up when the program ends or the original/backup files are updated.

Databases

When a program is using a database manager as its persistent storage choice, it is likely to have some of the following characteristics,

- Data items are sets, and individual items within the set are interacted with independently. For example, a set of employees, where individual employee contact information is updated when an employee moves.
- The program can be long running, in the case of a multi-user program it may be always running, so changes in the data will occur many times during the execution of the program.
- Multiple programs may interact with the same data items, requiring control procedures to be in place to prevent inconsistencies being introduced by conflicting interactions.
- Quantity of data requires use of indexes, storage optimizations, et cetera to meet the expected performance goals of the programs using it.
- Automation of backup, recovery, maintenance, and other administrative functions is desired.

When these characteristics are desired, a number of data management options are available. These include relational databases (using generic object storage options) or more specialized databases like *MongoDB* that support *JSON* as a native data type.

Robust File Storage

In between the two prior scenarios are many programs that have more robust requirements for robust data management, but do not have functional needs leading to selection of a database. Configuration files are a good example, in many cases they have robust auto-recovery requirements, while also requiring the easy manipulation afforded by file system based persistence.

The next topic will address the issues and options to address different aspects of persistence, providing choices and guidance on selecting and implementing robust data management in programs.

Recovery Enabled File Storage

If a program is able to automatically recover from faults, it will be considered more robust than a program that requires user/administrator intervention when any fault occurs. Most programs will not be able to recover from 100% of faults on their own - hardware limitations, cost of redundancy, and the need for additional decision-making criteria (e.g.,when the program has an error that has corrupted part of the data) all may limit getting to 100% automation. However, there are areas where the file management aspects can be made very robust, and contribute to getting programs closer to 100% automation in recovery processing.

There are classes of faults to consider. These include,

- Was the failure of the program itself due to an error in the software?
- Was the failure of the program due to an error caused by the hardware it is dependent on (e.g.,processor failure, network failure, storage failure)? Was the fault temporary, or does the program need to be relocated, or the hardware replaced?
- Timing of the failure. Did the the fault occurs at a point where data consistency could be an issue (e.g.,part way through writing data)?
- Impact of the failure. Is the data impacted user / administrator created/edited data or data managed by the program itself? Is the impacted data able to be created again from other sources? Does the user / administrator need to be involved in a recovery procedure to verify any part of the process or make decisions on recovery choices?

To provide robust software, to minimize the impact of these potential faults, design choices and implementation examples show how *JSON* files can be incorporated.

Failures, What the Recovery Process Needs to Know

When a program fails due to a software failure (the program terminates unexpectedly or is terminated by an external process / person), then the normal shutdown procedures cannot be assumed to have completed correctly.

When a hardware failure causes the software to fail completely (e.g., processor failure), then normal shutdown procedures will not have been processed, and even exceptional processing (such as an operating system signal) will not have been processed. Other hardware errors can result in blocking the software from completing a task, but allowing the software to recognize the fault and provide some mitigation. For example, a network failure that prevents access to the file system can provide an error message to the console / program user interface, even if the program does not have an alternate location to place the data to write, and other program shutdown procedures can be performed.

The key consideration is that the automated part of the recovery process can only rely on what is available after the fact in order to guide its processing. For example, it may not be able to determine whether a data update was partially completed or not, only that the file exists or doesn't exist that was to hold the data. Thus, the recovery processing needs to,

- Be able to determine that the permanent files related to the program were in a stable state.
- Be able to determine whether any temporary files related to the program were in a stable state.
- Know what automated steps to take for each of the files based on their state (stable or not) when the recovery process is initiated.
- Communicate whether the recovery steps are able to be performed through its automation, or whether additional intervention is required. Note, additional intervention could be additional automation (such as retrieval of a backup file from an automated backup facility or running a data consistency function), user action (such as inspection of data), or administrative action (such as providing a version of the data from a backup).

Some design choices are informed by these needs.

- For permanent files, their stable state must be able to be determined by examination of the file system, not by reliance on state information in the running program. If the program is to be installed in an environment where delayed write processing is possible, where an independent recovery process is not guaranteed to provide recovery coverage for writes in progress, then a mechanism for recognizing stable files is required.
- For temporary files, design choices include,
  - Removing temporary files as part of recovery processing, effectively deleting the data.
  - Placing the temporary files into a location for the user / administrator to access, leaving any further use of the files to the user / administrator.
  - Providing protection for the temporary files in the same manner as permanent files, and processing temporary files that can be verified as stable as part of recovery processing. For those that cannot be verified as stable, they can be handled in either of the manners described in the preceding choices.
- Simple processes occur only at the time of writing data (e.g.,create backup version). However, supporting recovery processing involves logic at the start of the program, for reads and writes, and potentially status interfaces that can be interacted with from within the program or with external processes. These multiple points of interaction may drive design choices related to modularization of logic, interface design, and access/security choices if external interactions are to be supported.

An example implementing a set of these design choices follows. From this example, many derivatives are possible, reflecting differing functional and non-functional requirements, environments, and amount of automation desired. The key concept is that *JSON* content can be utilized across a wide variety of programs and environments, schema definitions can be managed in the same manner, and incorporating robust handling in programs is feasible.

Library: safeFile

In the validation program in *chapter 8*, the *safeFile* module was referenced, being used the file reading function provided. In that example, the basic read function was used, rather than the recoverable interface, since the program required only read function and was not part of a larger set of programs that included read/write functionality.

In the upcoming example, the recoverable interface will be used, showing how a common module can be used to provide auto-recovery capabilities. To start, the *safeFile* module is presented, providing the context for the capabilities covered in its implementation.

- Synchronous functions are defined. This improves readability of the code, and for smaller files in local file systems is an appropriate design consideration.
- Both recovery enabled, and non-recovery enabled, read/write functions are provided. Programs may not require recovery automation for all files, but having a consistent interface is useful. This also allows subsequent versions of a program to switch between the two interface choices without changing libraries.

The design choices for the auto-recovery features in the module are,

- On write, the module will create a stable state in the file system that allows determining whether the write process completed or not by using a two stage write process. The first stage will write the data to an ephemeral file (extension *.eph*). After the file is closed, the second stage will rename the file to its ready state (extension *.rdy*). This will allow inspection of the file system state to determine whether the write process completed (existence of *.rdy* file), and the recovery process to recognize an incomplete processing of the 2 stages (existence of *.eph* file).
- The write processing will start with the creation of the *.eph* file to ensure the data is saved at the earliest time, without disrupting the previously stored data.
- The write steps for managing the transition of the existing backups and introduction of the new content to the permanent representations will be consistent with the auto recovery steps to perform the same activity.
- An interface will be provided that allows a program to ask for the status of the persistent data, indicating a ready status, auto-recovery possible status, or an intervention required status.
- An interface will be provided to initiate recovery processing.
- The read function will initiate auto-recovery if it is required.

Implementation descriptions for the *Javascript / Node.js* and *Python* versions follow.

Implementation of safeFile for Javascript / Node.js

index.js. As part of the packaging, the *Javascript / Node.js* version includes an *index.js* file that provides an exports directory for the library, since exports are available from multiple files.

Directory: chapter9/nodejs/safefile, file: index.js

```
module.exports.SafeFileError = require ("./SafeFileError").SafeFileError;
module.exports.safeFile = require ("./safeFile");
```

SafeFileError.js. A custom error definition is defined for the library, which can be presented through exception handling or in callbacks.

The *SafeFileError* declaration and implementation stores a message code and message. The values of the message code are provided through a set of constants (lines 18-26) that are attached to the *SafeFileError* prototype. Lines 28-32 define the message strings that are used with the format utility to construct the messages.

```
 1 /**
 2  * Error for safeFile errors/exceptions
 3  */
 4
 5 /**
 6  * Create SafeFileError instance with code and message.
 7  * @param {integer} code Message identifier.
 8  * @param [string] message Message text.
 9  */
10 function SafeFileError (code, message) {
11     this.name = "SafeFileError";
12     this.code = code;
13     this.message = message;
14 }
15
16 // constants
17 var p = SafeFileError.prototype;
18 p.NO_ERROR = 0;
19 p.INVALID_NAME = 100;
20 p.DOES_NOT_EXIST = 101;
21 p.IS_NOT_A_FILE = 102;
22 p.READ_ERROR = 103;
23 p.WRITE_ERROR = 104;
24 p.SAFE_NORMAL = 0;
25 p.SAFE_RECOVERABLE = 110;
26 p.SAFE_INTERVENE = 111;
27
28 p.MSG_INVALID_NAME = "File name missing or not valid";
29 p.MSG_IS_NOT_A_FILE = "File {0} is not a file";
30 p.MSG_DOES_NOT_EXIST = "File {0} does not exist";
31 p.MSG_READ_ERROR = "Error reading file {0}: {1}";
32 p.MSG_WRITE_ERROR = "Error writing file {0}: {1}";
33
34 //exports
35 exports.SafeFileError = SafeFileError;
```

safeFile.js. The implementation of *safeFile* has six exported functions providing access to normal read/write operations, auto-recoverable read/write operations, and recovery utility operations. The leading content includes the list of exported functions.

Directory: chapter9/node/safefile, file: safeFile.js

```
 1 /**
 2  * File processing functions for managed files.
 3  *
 4  * readFileSync - Read a file
 5  * writeFileSync - Write a file
 6  * safeGetState - Get the recovery state for a file
 7  * safeRecover - Initiate recovery for a file
 8  * safeReadFileSync - Read with recovery support
 9  * safeWriteFileSync - Write with recovery support
10  */
11 // import format function and Node.js file system module
12 var format = require ("ujs-format").format;
13 var fs = require ("fs");
14 var SafeFileError = require ("./SafeFileError").SafeFileError;
15 var cc = SafeFileError.prototype;
```

The *readFileSync* function implements a synchronous file read, throwing appropriate exceptions if any parameters are invalid or an error occurs. If no errors occur, the data from the file is returned.

```
17 /**
18  * Read file.
19  * @param {String} fileName File to read.
20  * @returns {String} Data read from file.
21  * @throws SafeFileError
22  */
23 function readFileSync (fileName, options) {
24     verifyFileName (fileName);
25 
26     var info = getFileInfo (fileName);
27     if (info.exists === false) {
28         var message1 = format (cc.MSG_DOES_NOT_EXIST, fileName);
29         throw new SafeFileError (cc.DOES_NOT_EXIST, message1);
30     }
31     if (info.isFile === false) {
32         var message2 = format (cc.MSG_IS_NOT_A_FILE, fileName);
33         throw new SafeFileError (cc.IS_NOT_A_FILE, message2);
34     }
35 
36     // read data file
37     var data = null;
38     try {
39         data = fs.readFileSync (fileName, options);
40     } catch (e) {
41         var message3 = format (cc.MSG_READ_ERROR, fileName, e.message);
42         throw new SafeFileError (cc.READ_ERROR, message3);
43     }
44 
45     // return data
46     return (data);
47 }
```

The *writeFileSync* function synchronously writes the provided content to the specified file name. If no data is provided, the file will be written as an empty file. If the file already exists, it is replaced.

```
49 /**
50  * Write data to a file.
51  * @param {String} fileName Name of file (path optional).
52  * @param {String} data Data to write.
53  * @throws SafeFileError
54  */
55 function writeFileSync (fileName, data, options) {
56     verifyFileName (fileName);
57 
58     var info = getFileInfo (fileName);
59     if ((info.exists === true) && (info.isFile === false)) {
60         var message1 = format (cc.MSG_IS_NOT_A_FILE, fileName);
61         throw new SafeFileError (cc.IS_NOT_A_FILE, message1);
62     }
63 
64     // if content undefined or null, set data to empty string
65     if ((data === undefined) || (data === null)) {
66         data = "";
67     }
68 
69     // write file content, throwing exception on error occurring
70     try {
71         fs.writeFileSync (fileName, data, options);
72     } catch (e) {
73         var message2 = format (cc.MSG_WRITE_ERROR, e.message);
74         throw new SafeFileError (cc.WRITE_ERROR, message2);
```

```
    }
}
```

The *readFileSync* and *writeFileSync* functions do not implement auto-recovery features. They are used for files that do not require the auto-recovery capabilities. However, their error messages are consistent with the *safeReadFileSync* and *safeWriteFileSync* functions, enabling easy transition between the interfaces.

The safeGetState function determines the persistent state of the file and its recoverable elements. The returned status indicates whether the state is stable, auto-recoverable, or not auto-recoverable.

```
/**
 * Get status of the file.
 * @param {String} file Name of base file.
 * @returns {Integer} SAFE_NORMAL, SAFE_AUTO_RECOVERABLE, SAFE_INTERVENE,
 *     INVALID_NAME, IS_NOT_A_FILE, or DOES_NOT_EXIST
 */
function safeGetState (fileName) {
    if ((fileName === undefined) || (fileName === null)) {
        return (cc.INVALID_NAME);
    }
    // if fileName exists, verify it is a file
    var info = getFileInfo (fileName);
    if ((info.exists) && (info.isFile === false)) {
        return (cc.IS_NOT_A_FILE);
    }

    var state = getState (fileName);
    return (state.status);
}
```

The *safeRecover* function provides an interface that allows the auto-recovery processing to be initiated.

```
/**
 * Initiate auto-recovery processing.
 * @param {String} fileName Name of base file
 * @throws SafeFileError
 */
function safeRecover (fileName) {
    verifyFileName (fileName);

    // if fileName exists, verify it is a file
    var info = getFileInfo (fileName);
    if ((info.exists) && (info.isFile === false)) {
        var message1 = format (cc.MSG_IS_NOT_A_FILE, fileName);
        throw new SafeFileError (cc.IS_NOT_A_FILE, message1);
    }

    // get state, if doesn't exist throw error
    var state = getState (fileName);
    if (state.status === cc.DOES_NOT_EXIST) {
        var message2 = format (cc.MSG_DOES_NOT_EXIST, fileName);
        throw new SafeFileError (cc.DOES_NOT_EXIST, message2);
    }

    performRecovery (state, true);
}
```

The *safeReadFileSync* function reads the contents from a file. However, it first determines whether auto-recovery processing is required, and if so, it initiates the recovery processing.

```
/**
 * Read a file, performing recovery processing if necessary.
 * @param {String} file File to read.
 * @returns {String} Data read from file.
```

```
 * @throws SafeFileError
 */
function safeReadFileSync (fileName, options) {
    verifyFileName (fileName);

    // if fileName exists, verify it is a file
    var info = getFileInfo (fileName);
    if ((info.exists) && (info.isFile === false)) {
        var message = format (cc.MSG_IS_NOT_A_FILE, fileName);
        throw new SafeFileError (cc.IS_NOT_A_FILE, message);
    }

    // get state, if auto-recovery required, perform recovery
    var state = getState (fileName);
    if (state.status === cc.SAFE_RECOVERABLE) {
        performRecovery (state, true);
    }

    // perform read on file
    return (readFileSync (fileName, options));
}
```

The *safeWriteFileSync* function creates the recovery enabled elements as it saves the content. It determines the recovery state, performs clean up and recovery preparation steps, writes the content, and completes recovery preparation.

```
/**
 * Write data to a file using a recoverable process.
 * @param {String} fileName Name of file (path optional).
 * @param {String} data Data to write.
 * @throws SafeFileError
 */
function safeWriteFileSync (fileName, data, options) {
    verifyFileName (fileName);

    // if fileName exists, verify it is a file
    var info = getFileInfo (fileName);
    if ((info.exists) && (info.isFile === false)) {
        var message = format (cc.MSG_IS_NOT_A_FILE, fileName);
        throw new SafeFileError (cc.IS_NOT_A_FILE, message);
    }

    // get current file system state, and auto-recover if necessary
    var state = getState (fileName);

    // store data in well defined ephemeral file to allow manual recovery
    // If file already exists, remove it (failed prior recovery).
    if (state.ephemeral.exists) {
        fs.unlinkSync (state.ephemeral.name);
    }

    writeFileSync (state.ephemeral.name, data, options);
    state.ephemeral.exists = true;

    // if ready state file already exists, recover prior state
    if (state.ready.exists) {
        performRecovery (state, false);
    }

    fs.renameSync (state.ephemeral.name, state.ready.name);

    // refresh state and process recovery to set file system state
    state = getState (fileName);
    performRecovery (state, true);
```

```
187 }
```

Non-exported support functions provide common processing. The *verifyFileName* function (lines 189-201) throws an error if the file name provided to any function is invalid.

```
189 /**
190  * Verify fileName parameter.
191  * @param {String} fileName Name of file to verify.
192  * @throws SafeFileError
193  */
194 function verifyFileName (fileName)
195 {
196     // if fileName undefined or null, throw exception
197     if ((fileName === undefined) || (fileName === null)) {
198         var message1 = format (cc.MSG_INVALID_NAME);
199         throw new SafeFileError (cc.INVALID_NAME, message1);
200     }
201 }
```

The *getState* function examines the persistent state of the recovery environment to determine what state (stable, auto-recoverable, not auto-recoverable) it is in.

```
203 /**
204  * Get state for file system entities.
205  * @param {String} file Base file name.
206  * @returns {Object} State object.
207  */
208 function getState (file) {
209     // collect state for all possible data and recovery files
210     var state = {};
211     state.ephemeral = getFileInfo (file + ".eph");
212     state.ready = getFileInfo (file + ".rdy");
213     state.base = getFileInfo (file);
214     state.backup = getFileInfo (file + ".bak");
215     state.tertiary = getFileInfo (file + ".bk2");
216 
217     if (state.ephemeral.exists) {
218         state.status = cc.SAFE_INTERVENE;
219     } else if ((state.ready.exists) || (state.tertiary.exists)) {
220         state.status = cc.SAFE_RECOVERABLE;
221     } else if (state.base.exists) {
222         state.status = cc.SAFE_NORMAL;
223     } else {
224         if (state.backup.exists) {
225             state.status = cc.SAFE_RECOVERABLE;
226         } else {
227             state.status = cc.DOES_NOT_EXIST;
228         }
229     }
230 
231     return (state);
232 }
```

The *getFileInfo* function provides the meta information for a file name. This information is used to determine validity of requests (e.g.,not allowing existing directories to be processed as files).

```
234 /**
235  * Get existence, file/directory info for a file.
236  * @param {String} fileName Name of file to get info for
237  * @returns {Object}
238  */
239 function getFileInfo (fileName) {
240     var result = {};
241     result.name = fileName;
242     result.exists = fs.existsSync (fileName);
```

```
243     if (result.exists) {
244         var stats = fs.statSync (fileName);
245         result.isFile = stats.isFile ();
246         result.isDirectory = stats.isDirectory ();
247     }
248     else {
249         result.isFile = false;
250         result.isDirectory = false;
251     }
252 
253     return (result);
254 }
```

The *performRecovery* function applies the auto-recovery processing logic to transform a recoverable state to a stable state.

```
256 /**
257  * Evaluate save state, initiating recovery if necessary.
258  * @param {Object} state State object with file names and existence flags
259  */
260 function performRecovery (state, removeEphemeral) {
261     // if ephemeral flag true, and ephemeral file exists, remove it
262     if ((removeEphemeral) && (state.ephemeral.exists)) {
263         fs.unlinkSync (state.ephemeral.name);
264     }
265 
266     // if only backups exist, restore from backup
267     var baseAvailable = state.base.exists || state.ready.exists;
268     if (baseAvailable === false) {
269         if (state.tertiary.exists) {
270             if (state.backup.exists) {
271                 fs.renameSync (state.backup.name, state.base.name);
272                 fs.renameSync (state.tertiary.name, state.backup.name);
273             } else {
274                 fs.renameSync (state.tertiary.name, state.base.name);
275             }
276         } else if (state.backup.exists) {
277             fs.renameSync (state.backup.name, state.base.name);
278         }
279 
280         return;
281     }
282 
283     // if tertiary state file exists, remove it
284     if (state.tertiary.exists) {
285         fs.unlinkSync (state.tertiary.name);
286     }
287 
288     // if ready state file exists, update ready, base and backup files
289     if (state.ready.exists) {
290         var removeTertiary = false;
291 
292 
293         // if base and backup exist, rename to tertiary temporarily
294         if ((state.base.exists) && (state.backup.exists)) {
295             fs.renameSync (state.backup.name, state.tertiary.name);
296             removeTertiary = true;
297         }
298 
299         // if base exists, rename to backup
300         if (state.base.exists) {
301             fs.renameSync (state.base.name, state.backup.name);
302         }
303 
```

```
304         // place ready state file in base and delete temporary tertiary file
305         fs.renameSync (state.ready.name, state.base.name);
306
307         // if temporary tertiary created, remove it
308         if (removeTertiary) {
309             fs.unlinkSync (state.tertiary.name);
310         }
311     }
312 }
```

The recovery processing includes a significant amount of file system manipulation in a deliberate set of steps. This enables the persistent state to be inspected after unexpected interruptions, and for recovery processing to progress irrespective of where the interruption occurred.

At the end of the file is the list of exports.

```
314 // exports
315 exports.readFileSync = readFileSync;
316 exports.writeFileSync = writeFileSync;
317 exports.safeGetState = safeGetState;
318 exports.safeRecover = safeRecover;
319 exports.safeReadFileSync = safeReadFileSync;
320 exports.safeWriteFileSync = safeWriteFileSync;
```

Format Utility for Javascript / Node.js

The *SafeFileError* message definitions use a replacement parameter syntax. *Javascript* does not have a built in function that provides exactly this function, so one is provided. The function is fairly simple, taking a base template string and a variable number of arguments, substituting the variable arguments where the template has substitution markers. Substitutions are indexed, so an argument can appear more than once in the template.

Directory: chapter9/nodejs/format, file: format.js

```
/**
 * Accept a base string with substitution markers and additional parameters
 * containing text to substitute into the base string. Markers use the syntax
 * {#} where # is a zero based index for the parameter ({0}, {1}, ...).
 *
 * @example
 * // returns "Syntax error on line 101: Missing ')'"
 * format ("Syntax error on line {0}: Missing '{1}'", "101", ")");
 *
 * @param {string} base Base string to substitute into
 * @param {...string} Substitution strings
 * @returns {string} Populated string
 */
function format (base)
{
    "use strict";
    // if base undefined or null, return empty string
    if ((base === undefined) || (base === null)) {
        return ("");
    }

    // for each argument after base, replace in base string
    var result = base;
    for (var ctr = 1; ctr < arguments.length; ctr ++) {
        result = result.replace ("{" + (ctr - 1) + "}", arguments[ctr]);
    }

    return (result);
}
```

```
// exports
exports.format = format;
```

Implementation of safeFile for Python

The Python implementation is a single file (plus the normal *__init__.py*).

Directory: chapter9/python/safefile, file: safefile.py

The leading content defines the constants and messages.

```
 1 """
 2 File processing functions for managed files.
 3
 4 readFile - Read a file
 5 writeFile - Write a file
 6 safeGetState - Get the recovery state for a file
 7 safeRecover - Initiate recovery for a file
 8 safeReadFile - Read with recovery support
 9 safeWriteFile - Write with recovery support
10 """
11 from os import unlink, rename
12 from os.path import exists, isfile, isdir
13
14 # error and message constants
15 NO_ERROR = 0
16 INVALID_NAME= 100
17 DOES_NOT_EXIST = 101
18 IS_NOT_A_FILE = 102
19 READ_ERROR = 103
20 WRITE_ERROR = 104
21 SAFE_NORMAL = 0
22 SAFE_RECOVERABLE = 110
23 SAFE_INTERVENE = 111
24
25 MSG_INVALID_NAME = "File name missing or not valid"
26 MSG_IS_NOT_A_FILE = "File {0} is not a file"
27 MSG_DOES_NOT_EXIST = "File {0} does not exist"
28 MSG_READ_ERROR = "Error reading file {0}: {1}"
29 MSG_WRITE_ERROR = "Error writing file {0}: {1}"
```

The *SafeFileError* class is a subclass of Exception, defining two class instance variables to hold the error code and error message.

```
31 class SafeFileError (Exception):
32     """
33     Error definition thrown when an error occurs.
34     Args
35         code: Error number
36         message: Text message, suitable for display
37     """
38     def __init__ (self, code, message):
39         super (SafeFileError, self).__init__ ()
40         self.code = code
41         self.message = message
```

The *readFile* function implements a synchronous file read, raising appropriate exceptions if any parameters are invalid or an error occurs. If no errors occur, the data from the file is returned.

```
43 def readFile (file):
44     """
45     Read file.
46     Args:
47       file (str): Path / file name of file to read.
48     Returns:
```

```
49        data (str): Data read.
50      Raises:
51        SafeFileError
52      """
53      if file is None:
54          raise SafeFileError (INVALID_NAME, MSG_INVALID_NAME)
55
56      info = _getFileInfo (file)
57      if not info["exists"]:
58          raise SafeFileError (DOES_NOT_EXIST, MSG_DOES_NOT_EXIST.format (file))
59
60      if not info["isFile"]:
61          raise SafeFileError (IS_NOT_A_FILE, MSG_IS_NOT_A_FILE.format (file))
62
63      # read data file
64      try:
65          data = open (file).read ()
66          return data
67      except IOError as e:
68          raise SafeFileError (READ_ERROR,
69              MSG_READ_ERROR.format (file, e.strerror))
```

The *writeFile* function synchronously writes the provided content to the specified file name. If no data is provided, the file will be written as an empty file. If the file already exists, it is replaced.

```
71 def writeFile (file, data):
72      """
73      Write file.
74      Args:
75          file (str): Path / file name to write.
76          data (str): Data to write.
77      Raises:
78          SafeFileError
79      """
80      if file is None:
81          raise SafeFileError (INVALID_NAME, MSG_INVALID_NAME)
82
83      info = _getFileInfo (file)
84      if info["exists"] and not info["isFile"]:
85          raise SafeFileError (IS_NOT_A_FILE, MSG_IS_NOT_A_FILE.format (file))
86
87      # read data file
88      try:
89          file = open (file, "w")
90          file.write (data)
91          file.close ()
92      except IOError as e:
93          raise SafeFileError (WRITE_ERROR,
94              MSG_WRITE_ERROR.format (file, e.strerror))
```

The *readFile* and *writeFile* functions do not implement auto-recovery features. They are used for files that do not require the auto-recovery capabilities. However, their error messages are consistent with the *safeReadFile* and *safeWriteFile* functions, enabling easy transition between the interfaces.

The *safeGetState* function determines the persistent state of the file and its recoverable elements. The returned status indicates whether the state is stable, auto-recoverable, or not auto-recoverable.

```
 96 def safeGetState (file):
 97      """
 98      Get the status of a file in the recovery context.
 99      Args:
100          file File to get status for.
101      Retuns:
102          State, can be an error or recovery state.
```

```
103     """
104     if file is None:
105         return INVALID_NAME
106 
107     info = _getFileInfo (file)
108     if info["exists"] and not info["isFile"]:
109         return IS_NOT_A_FILE
110 
111     state = _getState (file)
112     return state["status"]
```

The *safeRecover* function provides an interface that allows the auto-recovery processing to be initiated.

```
114 def safeRecover (file):
115     """
116     Initiate the recovery processing for a file.
117     Args:
118         file File to performing processing for.
119     Raises:
120         SafeFileError
121     """
122     if file is None:
123         raise SafeFileError (INVALID_NAME, MSG_INVALID_NAME)
124 
125     info = _getFileInfo (file)
126     if info["exists"] and not info["isFile"]:
127         raise SafeFileError (IS_NOT_A_FILE, MSG_IS_NOT_A_FILE.format (file))
128 
129     state = _getState (file)
130     if state["status"] == DOES_NOT_EXIST:
131         raise SafeFileError (DOES_NOT_EXIST, MSG_DOES_NOT_EXIST.format (file))
132 
133     _performRecovery (state, True)
```

The *safeReadFile* function reads the contents from a file. However, it first determines whether auto-recovery processing is required, and if so, it initiates the recovery processing.

```
135 def safeReadFile (file):
136     """
137     Read a file, applying recovery processing if necessary.
138     Args:
139         file File to read.
140     Returns:
141         Data read from file.
142     Raises:
143         SafeFileError
144     """
145     if file is None:
146         raise SafeFileError (INVALID_NAME, MSG_INVALID_NAME)
147 
148     info = _getFileInfo (file)
149     if info["exists"] and not info["isFile"]:
150         raise SafeFileError (IS_NOT_A_FILE, MSG_IS_NOT_A_FILE.format (file))
151 
152     state = _getState (file)
153     if state["status"] == SAFE_RECOVERABLE:
154         _performRecovery (state, True)
155 
156     readFile (file)
```

The *safeWriteFile* function creates the recovery enabled elements as it saves the content. It determines the recovery state, performs clean up and recovery preparation steps, writes the content, and completes recovery preparation.

```
158 def safeWriteFile (file, data):
```

```
159     """
160     Write data to a file, applying recovery enabling processing.
161     Args:
162         file File to write to.
163         data Data to write.
164     Raises:
165         SafeFileError
166     """
167     if file is None:
168         raise SafeFileError (INVALID_NAME, MSG_INVALID_NAME)
169 
170     info = _getFileInfo (file)
171     if info["exists"] and not info["isFile"]:
172         raise SafeFileError (IS_NOT_A_FILE, MSG_IS_NOT_A_FILE.format (file))
173 
174     # get current file system state, and auto-recover if necessary
175     state = _getState (file)
176 
177     # store data in well defined ephemeral file to allow manual recovery.
178     # If file already exists, remove it (failed prior recovery).
179     if state["ephemeral"]["exists"]:
180         unlink (state["ephemeral"]["name"])
181 
182     writeFile (state["ephemeral"]["name"], data)
183     state["ephemeral"]["exists"] = True
184 
185     # if ready state file already exists, recover prior state
186     if state["ready"]["exists"]:
187         _performRecovery (state, False)
188 
189     rename (state["ephemeral"]["name"], state["ready"]["name"])
190 
191     # refresh state and process recovery to set file system state
192     state = _getState (file)
193     _performRecovery (state, True)
```

The *getState* function examines the persistent state of the recovery environment to determine what state (stable, auto-recoverable, not auto-recoverable) it is in.

```
195 def _getState (file):
196     """
197     Get file state.
198     Args:
199         file File to get state for.
200     Returns:
201         State object containing list of recovery files and overall status.
202     """
203     state = {}
204     state["ephemeral"] = _getFileInfo (file + ".eph")
205     state["ready"] = _getFileInfo (file + ".rdy")
206     state["base"] = _getFileInfo (file)
207     state["backup"] = _getFileInfo (file + ".bak")
208     state["tertiary"] = _getFileInfo (file + ".bk2")
209 
210     if state["ephemeral"]["exists"]:
211         state["status"] = SAFE_INTERVENE
212     elif state["ready"]["exists"] or state["tertiary"]["exists"]:
213         state["status"] = SAFE_RECOVERABLE
214     elif state["base"]["exists"]:
215         state["status"] = SAFE_NORMAL
216     else:
217         if state["backup"]["exists"]:
218             state["status"] = SAFE_RECOVERABLE
219         else:
```

```
220             state["status"] = DOES_NOT_EXIST
221
222     return state
```

The *getFileInfo* function provides the meta information for a file name. This information is used to determine validity of requests (e.g., not allowing existing directories to be processed as files).

```
224 def _getFileInfo (file):
225     """
226     Get information for a file (name, exists, is a file or directory.
227     Args:
228         file File to get information for.
229     Returns:
230         Dict with file info (name, exists, isFile, isDirectory)
231     """
232     info = {}
233     info["name"] = file
234     info["exists"] = exists (file)
235     info["isFile"] = isfile (file)
236     info["isDirectory"] = isdir (file)
237     return info
```

The *performRecovery* function applies the auto-recovery processing logic to transform a recoverable state to a stable state.

```
239 def _performRecovery (state, removeEphemeral):
240     """
241     Initiate recovery processing.
242     Args:
243         state State object with recovery file information.
244         removeEphemeral Flag, remove ephemeral if found or not
245     `"""
246     # if ephemeral flag true, and ephemeral file exists, remove it
247     if removeEphemeral and state["ephemeral"]["exists"]:
248         unlink (state["ephemeral"]["name"])
249
250     # if only backups exist, restore from backup
251     baseAvailable = state["base"]["exists"] or state["ready"]["exists"]
252     if not baseAvailable:
253         if state["tertiary"]["exists"]:
254             if state["backup"]["exists"]:
255                 rename (state["backup"]["name"], state["base"]["name"])
256                 rename (state["tertiary"]["name"], state["backup"]["name"])
257             else:
258                 rename (state["tertiary"]["name"], state["base"]["name"])
259         elif state["backup"]["exists"]:
260             rename (state["backup"]["name"], state["base"]["name"])
261         return
262
263     # if tertiary state file exists, remove it
264     if state["tertiary"]["exists"]:
265         unlink (state["tertiary"]["name"])
266
267     # if ready state file exists, update ready, base and backup files
268     if state["ready"]["exists"]:
269         removeTertiary = False
270
271         # if base and backup exist, rename to tertiary temporarily
272         if state["base"]["exists"] and state["backup"]["exists"]:
273             rename (state["backup"]["name"], state["tertiary"]["name"])
274             removeTertiary = True
275
276         # if base exists, rename to backup
277         if state["base"]["exists"]:
278             rename (state["base"]["name"], state["backup"]["name"])
```

```
279
280         # place ready state file in base and delete temporary tertiary file
281         rename (state["ready"]["name"], state["base"]["name"])
282
283         # if temporary tertiary created, remove it
284         if removeTertiary:
285             unlink (state["tertiary"]["name"])
```

The recovery processing includes a significant amount of file system manipulation in a deliberate set of steps. This enables the persistent state to be inspected after unexpected interruptions, and for recovery processing to progress irrespective of where the interruption occurred.

Test Cases for safeFile

A hardware failure, power failure, or other failure could occur at any point during the processing of the file management activities. This can leave a wide variety of states possible. To ensure the different states are properly recognized and handled, test suites are provided for each of the *safeFile* functions. Each test creates the files representing the starting state, calls the function to be tested, verifies that all possible state files are present/not-present as applicable to the test case, and cleans up after completing the test.

The test cases are not detailed here, but are available as part of the project content for both *Javascript / Node.js* (using *mocha*) and *Python* (using *pytest*) on *GitHub*, and are executed using automated testing.

Example: Using Robust Configuration File Capabilities

A skeleton program is used for the example, showing the following activities.

- Determine the state of its configuration file at the start of the program. This provides the opportunity to direct user action if needed.
- Load the configuration.
- Make changes to the configuration.
- Write the configuration.

The program logic reads a simple inventory file, updates the inventory count for each of the items, and writes out the updated content.

Implementation Using Javascript / Node.js

The leading content sets the variables and displays a starting message. Lines 17-28 check the persistent state, and some typical courses of action are shown. Lines 30-42 show the loading and validation processing, including error handling for invalid content. Lines 44-48 provide placeholder logic, representing the program content. Lines 50-59 show preparing and saving the updated content. Finally, line 61 displays an end message.

Directory: chapter9, file: inventory.js

```
1 /*
2  * Inventory management simulation
3  */
4 var safeFile = require ("ujs-safefile").safeFile;
5 var SafeFileError = require ("ujs-safefile").SafeFileError;
6 var cc = SafeFileError.prototype;
7 var jsonValidate = require ("ujs-jsonvalidate");
8 var validate = jsonValidate.validate;
9
```

```
10 // starting message
11 console.log ("Starting processing");
12
13 // inventory files
14 var dataFile = "inventory.json";
15 var schema = "inventory_schema.json";
16
17 // determine current state
18 var status = safeFile.safeGetState (dataFile);
19 if (status === cc.SAFE_INTERVENE) {
20     console.log ("Inventory file requires administrator action");
21     process.exit (1);
22 } else if (status === cc.DOES_NOT_EXIST) {
23     console.log ("Inventory file missing");
24     process.exit (1);
25 } else if (status === cc.SAFE_RECOVERABLE) {
26     safeFile.safeRecover (dataFile);
27     console.log ("Inventory file auto recovered");
28 }
29
30 // load and validate file content
31 var inventory = null;
32 validate (dataFile, schema, null, null, function (code, data, message) {
33     // if invalid, print error message
34     if (code !== jsonValidate.VALID) {
35         console.log ("Inventory file validation failed");
36         console.log ("Error: " + message);
37         process.exit (1);
38     }
39
40     // assign content
41     inventory = data;
42 });
43
44 // program content goes here ...
45 // to show updates, increment item count by 1 for all items
46 for (var ctr = 0; ctr < inventory.length; ctr ++) {
47     inventory[ctr].count = inventory[ctr].count + 1;
48 }
49
50 // apply formatting to content before writing when
51 // content is intended to be user readable/editable
52 var output = JSON.stringify (inventory, null, 2);
53
54 // write using recoverable interface
55 try {
56     safeFile.safeWriteFileSync (dataFile, output);
57 } catch (e) {
58     console.log ("Error writing content " + e.message);
59 }
60
61 console.log ("Completed processing");
```

Lines 50-52 are relevant when the *JSON* content is intended for user reading or editing. The *stringify* function will separate the content into lines and provide indenting. If this is not used, then the content will be emitted using the normal *JSON* serializer which creates a continuous output. Either form can be read by the *JSON* parser.

This program can be modified to try out a variety of conditions by updating the data file and/or changing the *dataFile* and *schema* variables on lines 14-15 to reference different files. Using the unit test

cases as examples (*test/test-safeFile.js* from the *ujs-safefile-nodejs* repository), different persistent configurations can be tested.

Implementation using Python

The leading content sets the variables and displays a starting message. Lines 12-26 check the persistent state, and some typical courses of action are shown. Lines 28-35 show the loading and validation processing, including error handling for invalid content. Lines 37-40 provide placeholder logic, representing the program content. Lines 42-50 show preparing and saving the updated content. Finally, line 52 displays an end message.

Directory: chapter9, file: inventory.py

```
 1 """
 2 Inventory management simulation
 3 """
 4 import sys
 5 from safefile import safefile, SafeFileError
 6 from jsonvalidate import validate, VALID
 7 from json import dumps
 8
 9 # starting message
10 print ("Starting processing")
11
12 # inventory files
13 dataFile = "inventory.json";
14 schema = "inventory_schema.json";
15
16 # determine current state
17 status = safefile.safeGetState (dataFile);
18 if status == safefile.SAFE_INTERVENE:
19     print ("Inventory file requires administrator action")
20     sys.exit (1)
21 elif status == safefile.DOES_NOT_EXIST:
22     print ("Inventory file missing")
23     sys.exit (1)
24 elif status == safefile.SAFE_RECOVERABLE:
25     safefile.safeRecover (dataFile)
26     print ("Inventory file auto recovered")
27
28 # load and validate file content
29 inventory = None;
30 code, inventory, message = validate (dataFile, schema, None, None)
31 # if invalid, print error message
32 if code != VALID:
33     print ("Inventory file validation failed")
34     print ("Error: " + message)
35     sys.exit (1)
36
37 # program content goes here ...
38 # to show updates, increment item count by 1 for all items
39 for item in inventory:
40     item['count'] = item['count'] + 1
41
42 # apply formatting to content before writing when
43 # content is intended to be user readable/editable
44 output = dumps (inventory, indent = 2, sort_keys = True);
45
46 # write using recoverable interface
47 try:
48     safefile.safeWriteFile (dataFile, output)
49 except SafeFileError as e:
```

```
50     print ("Error writing content " + e.message)
51
52 print ("Completed processing")
```

Lines 42-44 are relevant when the *JSON* content is intended for user reading or editing. The indent argument specified on the dumps function will direct separation of the content onto lines and provide indenting. If this is not used, then the content will be emitted using the normal *JSON* serializer which creates a continuous output. Either form can be read by the *JSON* parser.

This program can be modified to try out a variety of conditions by updating the data file and/or changing the *dataFile* and *schema* variables on lines 13-14 to reference different files. Using the unit test cases as examples (*test/test-safefile.py* from the *ujs-safefile-python* repository), different persistent configurations can be tested.

Multiple Data File Programs

In the organization / employee example, the data was stored in two separate files. The *safeFile* library implementation included with the book materials can be used to manage multiple files, but as independent files, not as a coordinated set of files.

To support managing a set of files as a coordinated unit, meaning storage and recovery apply to the set as a whole rather than as independent files, additional features need to be considered, including,

- The ephemeral state for the complete set of coordinated files needs to be established before any changes to the set of existing files are applied.
- The final state for all files in the coordinated set needs to be the same for all files in the set. This allows subsequent inspection to be based on a stable inspection criteria. For example, each file will have a base file, and all or none will have a backup file (*base* + *.bak*).
- The reassignment of existing files to different stages must be performed in a manner that can be recognized by the recovery process. This will allow the recovery process to perform automatic recovery where the complete process did not complete.
- A failure of writing for one ephemeral file will be considered a failure for the group of coordinated files. A design consideration would be the addition of an interim state between ephemeral and ready, applicable for recognizing each individual file as being in the ready state, where the coordinated set may not be. This would be of assistance to manual inspection, or auto-recovery scenarios where some ephemeral files can be regenerated.

The considerations for the different interaction models (read at start / write at end, user / program directed, auto-save) apply to the multiple file design as well.

Appendix A: Installing the Book Materials

The examples , sample source code, and tools are available to download and access online.

To download the examples and source code for all programs, download the package for your operating system from the *Using JSON Schema* website,

http://usingjsonschema.com

For *Linux*, *OS X* and *Unix* variants, the package is a compressed tar file. Download the package, and assuming the default download location is used (*~/Downloads*), then in a *Terminal* use the following commands. The following command places the content in a sub-directory of your home directory.

```
cd ~
gzip xvf Downloads/usingjsonschema.gzip
```

On Windows, download the package. by default, it will be placed in the directory

%HOMEDRIVE%%HOMEPATH%\Downloads

where *%HOMEDRIVE%%HOMEPATH%* are similar to *C:\Users\Fred*. You can make the package a sub-directory of your home directory, or place it elsewhere.

```
cd %HOMEDRIVE%%HOMEPATH%
unzip Downloads\usingjsonschema.zip
```

The directory structure for the contents will be created as part of the package contents being extracted.

Installing the Syntax and Validation Tools

These tools are installed from the *Node.js* and *Python* repositories, using the package management utilities for each. You can choose to install either one, or install both.

For *Node.js*, if the *Node.js* runtime platform is not already installed, see *Appendix C* to install it. When installed, use the following commands to install the *jsonsyntax* and *validate* tools.

```
npm -g install ujs-jsonsyntax
npm -g install ujs-jsonvalidate
```

The *-g* parameter instructs *npm* to install the tools into the global installation location, allowing the tools to be available when used in any directory on the system.

For *Python*, if the *Python* platform is not already installed, see *Appendix D* to install it. When installed, use the following commands to install the *jsonsyntax* and *validate* tools.

```
pip install ujs-jsonsyntax
pip install ujs-jsonvalidate
```

For versions of *Python* before 3.3, you may have to add the scripts directory under your *Python* installation directory to the system path to use the commands from any directory.

Using Git to Access the Projects

All of the source code and examples are available from *GitHub* under the *Using JSON Schema* project at,

https://github.com/usingjsonschema

A project navigation page is provided at,

https://usingjsonschema.github.io

The source code can be accessed directly from each repository web page, or *git clone* can be used to create a local instance in your own *git* repository. A zipped version of the repository can also be downloaded using the *Download ZIP* button.

The *GitHub* content includes development content, including unit tests, that will be helpful for experimentation with the projects. The content can be used with either command line or integrated development environment (IDE) tools.

Appendix B: Resources

JSON Schema Resources

The *JSON Schema* home website is at *http://json-schema.org* and contains descriptions of the technologies and links to various resources.

The format for *JSON* content is published as *RFC 7159* by the *IETF* (*Internet Engineering Task Force*) and is available at

http://tools.ietf.org/html/rfc7159

The *JSON Schema* core and validation draft specifications are also available through the *IETF* at,

Core specification

http://tools.ietf.org/html/draft-zyp-json-schema-04

Validation specification

http://tools.ietf.org/html/draft-fge-json-schema-validation-00

A message group for *JSON Schema* use, direction, and general discussion, is available at

https://groups.google.com/forum/#!forum/json-schema

ECMAScript Resources

The *ECMAScript* specification (*ECMA 262*) is available at

http://www.ecma-international.org/publications/standards/Ecma-262.htm

The specification includes the regular expression definitions used by *JSON Schema*.

Postal Address Resources

Canadian provinces and territories, from *Canada Post*

http://www.canadapost.ca/tools/pg/manual/PGaddress-e.asp#1382088

USA states and territories, from the *US Postal Service*

https://www.usps.com/send/official-abbreaviations.html

Mexico states, from the *Universal Postal Union*

http://www.upu.int/fileadmin/documentsFiles/activities/addressingUnit/mexEn.pdf

Appendix C: Node.js Installation and Introduction

Node.js is a runtime platform that includes the *V8 Javascript Engine*.

- The *Node.js* project is an open source project, with project management led by *Joyent*.
- The *V8 Javascript Engine* project is an open source project, with project management led by *Google*.

The examples use *Node.js* to provide access to the *Javascript* engine as well as the modules packaged with *Node.js* for file system and networking functions.

There is no cost for the use of either of these technologies for the examples used in this book.

Node.js Installation

Node.js is available for many platforms. Short instructions are provided for some common platforms here. Full instructions for all platforms is available at,

https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

Node.js Install (Ubuntu)

For *Ubuntu 14.04* (April 2014 release), the *Ubuntu* repository contains a 0.10.x release of *Node.js* which can be used for the book software. To install, use the following commands.

```
sudo apt-get update
sudo apt-get -y install nodejs
sudo apt-get -y install npm
```

For earlier versions of *Ubuntu*, to install the current stable release of *Node.js* software, open a *Terminal* window and use the following commands,

```
sudo apt-get update
sudo apt-get -y install python-software-properties
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get -y install nodejs
```

The *update* command updates the package manager catalog.

The *python-software-properties* package supports the *add-apt-repository* command.

The *add-apt-repository* command allows external package management repositories to be accessed. In this case the command adds the *Node.js* distribution repository to the *apt-get* directory.

The second *update* command updates the catalog to include the new repository information.

The final command installs *Node.js*.

Node.js Install (Fedora)

To install the *Node.js* software on *Fedora*, open a *Terminal* window and use the following commands,

```
sudo yum install nodejs npm
```

Node.js Install (Windows)

To acquire the *Node.js* software for *Windows*, open a web browser and go to

http://nodejs.org/download

This is the primary *Node.js* download location. In the table of choices, on the *Windows Installer (.msi)* row, select the appropriate download for your system (32-bit or 64-bit). Save the installer on your system and run it after the download completes. The install setup wizard will guide the process of installing the runtime.

- The *End-User License Agreement* page includes the license information for *Node.js* and its dependencies.
- The *Destination Folder* page allows setting the install location, the default is suitable.
- The *Custom Setup* page allows selection of install content. Leave all items selected.
- When the *Ready to install Node.js* page is displayed, press *Install*.

At the completion of the installation, the *Node.js* runtime and *Node Package Manager* (*npm*) will be ready to use.

Introduction to Node.js

For readers not familiar with *Node.js*, the following will provide some basic information to assist in reading the *Javascript* versions of the programs presented in this book.

Node.js packages up a set of *Javascript* extensions and provides a runtime library for a variety of functions. Some of the constructs, like *modules* and *require*, can also be used outside *Node.js*, but are inherent parts of any non-trivial *Node.js* program.

Asynchronous Programming

Node.js supports both synchronous and asynchronous programming for many functions, however some functions are asynchronous only. This is part of the overall architecture of *Node.js*, and how it enables multiple tasks to be worked on simultaneously. A common pattern to replace the traditional,

```
function someFunction (arg) { return (0); }

result = someFunction (arg);
if (result === 0) { console.log ("ok");
```

is to use a callback function that receives the result in the callback.

```
function someFunction (arg) { callback (0); }

function callback (result) {
    if (result === 0) { console.log ("ok");
}
someFunction (arg, callback);
```

Alternatively, and very commonly, the callback function will be part of the call.

```
function someFunction (arg) { callback (0); }

someFunction (arg, function callback (result) {
    if (result === 0) { console.log ("ok");
});
```

Asynchronous programming logic can be nested to ensure that tasks occur in a sequential manner, or the tasks can be run in parallel and coordinated as a group to do processing only after all tasks have been completed. An example of the latter of these cases is shown in the httpFetch function in the *validate* module from the *ujs-json-validate* project.

Modules

Node.js uses a module concept. Global variables, those not declared with a *var* statement, are not used in the any of the programs in this book. If a variable is to be shared outside its module, it will use the *Node.js* exports mechanism.

Require

The *require* function used at the start of the programs is part of the module support of *Node.js*. It is similar to import or include statements from other languages. They include the content of another module for use in this program. For user supplied modules, the path to the module is included (e.g., "./mymodule"). Modules included from the *Node.js* platform or from other sources use only the name (e.g., "http"). This distinction provides the runtime with direction on how to search for the module.

Typically the *require* statement will be part of a *var* declaration. When a module is resolved, the result of the resolution can be assigned so that subsequent interactions can be performed. Two common patterns are seen.

```
var http = require ("http");
```

This form assigns the module to the variable *http*. When the program want to call a function supplied by the *http* module, is can use syntax *http.request (…)* to initiate the call. Note the variable can be any name, but typically the variable name and module name will be similar in cases like these.

The second pattern assigns an element within the module. For example,

```
var format = require ("./format").format;
```

In this case, the variable *format* is being assigned the format function from the module. This syntax allows subsequent use of statements such as,

```
format (base, 100);
```

In contrast, if just the module was assigned, then the qualifier would be required.

```
format.format (base, 100);
```

Exports

Using *exports* (and *module.exports*) enables elements to be made visible outside the module they are defined in. This can be applied to both functions and variables. Any element that is not included in the exports object is private to the module.

Appendix D: Python Installation

Python is available in two versions, 2.x and 3.x. Either version can be used with the software and examples provided in this book.

Linux and OS X

For *Linux* and *OS X*, *Python* is installed as part of the base operating system for most distributions. However, additional versions can be installed alongside the version provided. Some distributions include version 2 and 3 installed, using *python* as the command for version 2.x.x and *python3* as the command for version 3.x.x. You can use the following commands to determine the installed versions.

```
python –version
python3 --version
```

All Platforms

If your *Linux* or *OS X* version is an older release, or if your operating system does not have *Python* already installed, instructions and downloads are available at,

https://www.python.org/downloads/

For P*ython 2*, version 2.7 or later is recommended.

For *Python 3*, version 3.4 or later is recommended (as this version bundles the *pip* package manager).

Python Package Management

Installing and managing *Python* packages is performed using the *pip* program. General information on *pip* is available at

http://pip.readthedocs.org/en/latest/index.html#

If *Python 3.4.0* or later is used, the installation information that follows can be ignored, since the *pip* package manager is installed with the base installation.

If using *Python 2* or a version of *Python 3* earlier than 3.4, then *pip* is a separate installation step. To install *pip*, see the installation page

http://pip.readthedocs.org/en/latest/installing.html

The *pip* package manager is the successor to *easy-install*. If the installation instructions for a package include use of *easy-install* then substitute *pip* where *easy-install* is referenced.

About the Author

Joe McIntyre is a computing enthusiast and software architect with 25 years of experience, working in the computing, telecommunications, and public utility industries.

Senior Technical Staff Member at IBM Corporation, Architecture Leader for the Communications Sector Industry Frameworks. Responsible for service delivery platform (SDP) architecture, he led the Cloud Services Provider solution architecture definition for the IBM Cloud Computing Reference Architecture. Previously at IBM, Joe was Chief Architect for the WebSphere family of products for the telecommunications industry and prior projects included distributed systems management, virtual desktop infrastructure, Java/JVM technologies, and object-oriented tooling/code generation technologies. He has approximately 75 issued patents.

Joe has also been active contributor and leader in standards activities, including representing IBM to the United States National Body for the International Standards Organization (ISO) in the Cloud Computing activity under JTC1/SC38. He was the editor of the 3GPP/ETSI Parlay X Web Services specifications, and a primary contributor to 3GPP, ETSI and Open Mobile Alliance standards activities for Web Services. He chaired the Web Services Working Group within The Parlay Group, and was convenor for the 3GPP CT5 (Core Networking and Terminal Group working group 5).

Joe graduated from Fanshawe College in Management Information Systems, and currently resides in Round Rock, Texas, USA.

Other Publications by the Author

Author of the IBM Redguide, *"IBM SmartCloud: Becoming a Cloud Services Provider"* (ISBN 9780738438054), published in December 2012 and available as a PDF or EPUB download at *http://www.redbooks.ibm.com/abstracts/redp4912.html* and from the Apple iBookstore.

Co-author, *"Extending the Service Bus for Successful and Sustainable IPTV Services"*, IEEE Communications Magazine (Volume 48, Issue 8), August 2008.

Foreword, *"JavaBeans for Dummies"* (ISBN 0764501534), published 1997 by IDG Books Worldwide Inc.