JSON Schema Basics

Kuda Dube
Computer Science and Information Technology
School of Fundamental Sciences

08/10/2019

Acknowledgement

This topic is based on material in Chapter 5 of the book:

Tom Marrs (2017). JSON at Work: Practical Data Integration for the Web. Chapter 5: pp. 113 - 158. O'Reilly Media, Inc. ISBN: 978-1-449-35832-7

About this Presentation

- This is an R Markdown presentation.
- Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.
- For more details on using R Markdown see http://rmarkdown.rstudio.com.

Topic Aim

To show and demonstrate how to leverage JSON Schema to:

- define the structure of JSON documents;
- define format of JSON documents; and
- validate JSON documents exchanged between applications.

Learning Objectives

At the end of this topic, you should be able to:

1. Explain the concepts of JSON Schema;
2. Use the Core JSON Schema and tools to define structure and format of JSON documents; and
3. Use JSON Schema to validate JSON documents using appropriate tools.

Software Requirements

- Practical work in this topic will require the following:
  - *JavaScript Engine: e.g., Node.js;*
  - *An Editor; and*
  - *A modern browser.*
- More information will be provided in tutorial and lab sessions.

Overview of JSON Schema

- A JSON Schema *specifies* a JSON document (or message)'s *content, structure,* and *format.*

- A JSON Schema also *validates* a JSON document — plain JSON validation isn't enough:

  - ***Syntactic validation*** — *validation only the syntax or well-formdedness of the document and does not require a schema. Tools for syntactic validation include: JSONLint and JSON parsers on various platforms;*

  - ***Semantic validation*** — *validating the meaning of the data not just the syntax and this requires the JSON Schema*

- JSON Schema is also useful in *API Design* because it helps to define the interface.

When do you need a JSON Schema?

- In situations when you need to validate at a deeper level by using semantic validation;
- Consider the following situations:
    1. You (as an API Consumer) need to ensure that a JSON response from an API contains a valid Speaker, or a list of Orders?
    2. You (as an API Producer) need to check incoming JSON to make sure that the Consumer can send you only the fields you're expecting?
    3. You need to check the format of a phone number, a date/time, a postal code, an email address, or a credit card number?

Example 1: A Basic Schema

The following JSON Schema example specifies that a document can have three fields (*email*, *firstName*, and *lastName*), where each one is a *string*.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    }
  }
}
```

Note that the JSON Schema is written in JSON itself just as the XML Schema is written in XML.

Example 2: JSON Document instance of Schema in Example 1

This example shows a sample JSON instance document that corresponds to the preceding Schema.

```
{
  "email": "jacinda@adern.com",
  "firstName": "Jacinda",
  "lastName": "Adern"
}
```

Key references:

- The online home of JSON Schema;
- The json-schema GitHub repository

Why JSON Schema?

- JSON Schema allows you to *validate content and semantics of a JSON document.*
- There are many situations and real-world cases where the Schema becomes a must. Examples include:
  - ***Security** — the OWASP recommends that Web Services should validate their payloads using a Schema.*
  - ***Message Design** — many enterprises use JSON as the preferred format to send payloads over messaging systems such as Apache Kafka.*
  - ***API Design** — JSON Schema helps define an API's contract by specifying the format, content, and structure of a document.*
  - ***Prototyping** — a streamlined prototyping workflow can be used with JSON Schema and related tooling when developers design an API.*

The JSON Schema Standard

- 16 September 2019: JSON Schema Specification is at implementation Draft 2019-09 (formerly known as draft-08);
- JSON Schema is robust, provides solid validation capabilities today;
- There is a wide variety of working *JSON Schema libraries* for every major programming platform.

JSON Schema vs XML Schema

- JSON Schema serves the same role with JSON as XML Schema does with XML documents;
- Key differences include:
  - *A JSON document does not reference a JSON Schema. It's up to an application to validate a JSON document against a Schema.*
  - *JSON Schemas have no namespace.*
  - *JSON Schema files have a* ***.json*** *extension (the same as for JSON documents).*

Developing JSON Schemas: Tools and Workflow

Key JSON Schema development tools include:

- JSON Editor Online (JOE);
- JSONSchema.net;
- JSON Validate;
- NPM Modules on the command line intergface (CLI): validate and jsonlint

Core Keywords in JSON Schema Langauge

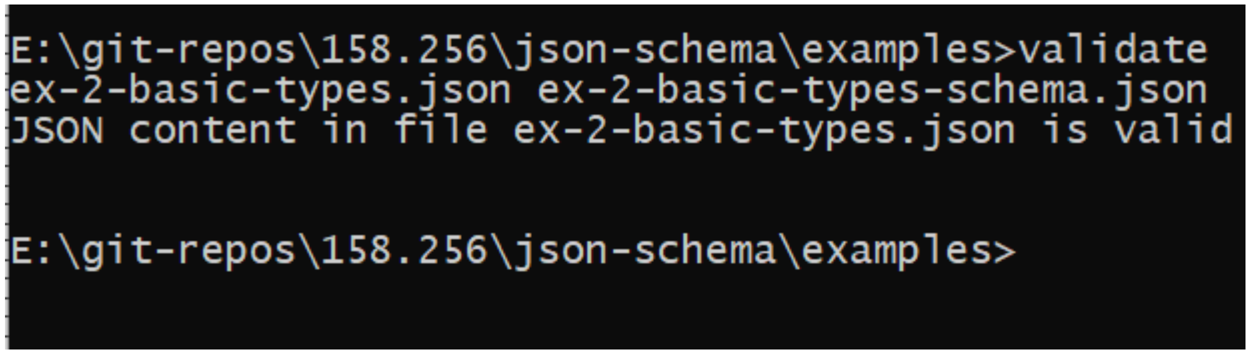
- **$schema** — specifies the JSON Schema (spec) version. For example, “`$schema": "http:// json-schema.org/draft-08/schema#"` specifies that the schema conforms to version 0.8, while `http://json-schema.org/schema#` tells a JSON Validator to use the current/latest version of the specification (which is 0.8 as of preparation of these slides).
- **type** — specifies the data type for a field. For example: “type”: “string”.
- **properties** — specifies the fields for an object. It also contains data type information.

Example 3: Basic Types in JSON Schema

The following **JSON document** contains the *basic JSON types* (for example, *string, number, boolean*):

```
{
  "email": "kuda@rukanda.com",
  "firstName": "Kuda",
  "lastName": "Dube",
  "age": 39,
  "postedSlides": true,
  "rating": 4.1
}
```

The **JSON Schema** on the right on this slide describes the structure of the above JSON document.



```
E:\git-repos\158.256\json-schema\examples>validate
ex-2-basic-types.json ex-2-basic-types-schema.json
JSON content in file ex-2-basic-types.json is valid

E:\git-repos\158.256\json-schema\examples>
```

Validating of document against the schema using the CLI

```
{
  "$schema":
  "URI:draf-08/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "age": {
      "type": "integer"
    },
    "postedSlides": {
      "type": "boolean"
    },
    "rating": {
      "type": "number"
    }
  }
}
```

Example 3: Remarks

In this example, note the following:

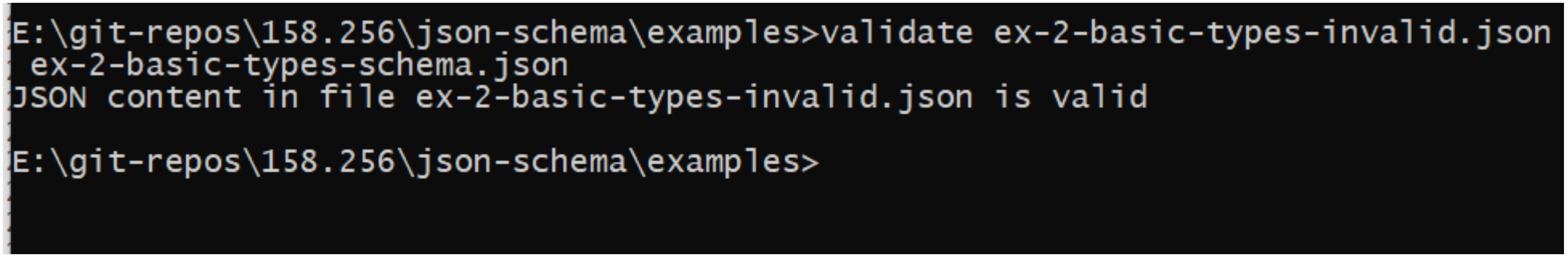
- The **$schema** field indicates that JSON Schema v0.8 rules will be used for validating the document;
- The first type field mentioned indicates that there is an Object at the root level of the JSON document that contains all the fields in the document;
- **email**, **firstName, lastName are of type string;
- age is an integer. Although JSON itself has only a number type, JSON Schema provides the finer-grained integer type.
- postedSlides is a boolean.
- rating is a number, which allows for floating-point values.

Example 4: An Invalid JSON Document

The following JSON document results from the following done to the previous JSON document: 1.Add an extra field (e.g., company). 2.Remove one of the expected fields (e.g., postedSlides).

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "age": 39,
  "rating": 4.1,
  "company": "None"
}
```

However, the following screenshot shows that this document is valid, which is surprising because JSON Schema is expected to validate as expected.



```
E:\git-repos\158.256\json-schema\examples>validate ex-2-basic-types-invalid.json
 ex-2-basic-types-schema.json
JSON content in file ex-2-basic-types-invalid.json is valid

E:\git-repos\158.256\json-schema\examples>
```

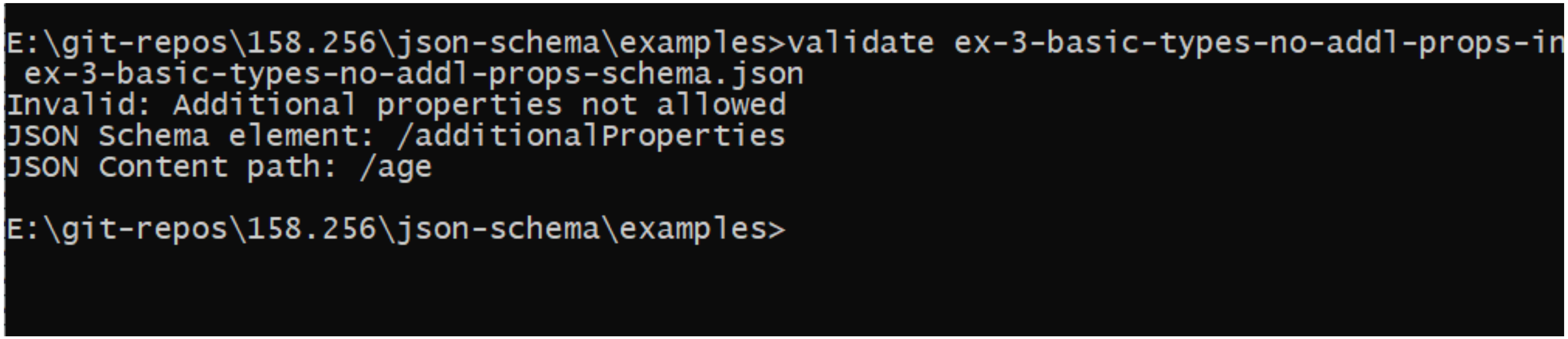
Example case where validation by JSON Schema seem to fail

Example 5: Basic types validation — adding simple constraints

It's possible to make the validation process function as expected by adding simple constraints to prevent extra fields. The the code on the right of this slid illustrates this.

- In this example, setting additionalProperties to false disallows any extra fields in the document root Object.
- Copy the previous JSON document (`ex-2-basic-typesinvalid.json`) to a new version (`ex-3-basic-types-no-addl-props-invalid.json`) and try validating against the preceding Schema.
- You should now see the following:

```
{
  "$schema":
    "URI/draft-08/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "postedSlides": {
      "type": "boolean"
    },
    "rating": {
      "type": "number"
    }
  },
  "additionalProperties": false
}
```



```
E:\git-repos\158.256\json-schema\examples>validate ex-3-basic-types-no-addl-props-in
 ex-3-basic-types-no-addl-props-schema.json
Invalid: Additional properties not allowed
JSON Schema element: /additionalProperties
JSON Content path: /age

E:\git-repos\158.256\json-schema\examples>
```

Example case where validation by JSON Schema succeeds

Example 6: Numbers, Compulsory and Optional Fields

- To reach a core level of semantic validation, you need to ensure that all required fields are present, as shown on the right.
- The Schema validates the average **rating** for a

```
{
  "$schema": "URI/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "postedSlides": {
```

speaker's conference presentation, where the range varies from 1.0 (poor) to 5.0 (excellent).

- In this example, the **required** Array specifies the fields that are required, so these fields must be present for a document to be considered valid.
- Note that a field is considered optional if not mentioned in the **required** Array.

```
        "type": "boolean"
      },
      "rating": {
        "type": "number",
        "minimum": 1.0,
        "maximum": 5.0
      }
    },
    "additionalProperties": false,
    "required": ["email", "firstName",
      "lastName", "postedSlides", "rating"]
  }
```

Example 7: Arrays in JSON Schema

- Arrays can hold any of the JSON Schema basic types (*string, number, array, object, boolean, null*).

- The Schema in the following example validates the tags field, which is an Array of type string.
- JSON Schema provides the ability to specify the minimum (minItems) and maximum (maxItems) number of items in an Array.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
      "tags": {
        "type": "array",
        "minItems": 2,
        "maxItems": 4,
        "items": {
          "type": "string"
        }
    }
  },
  "additionalProperties": false,
  "required": ["tags"]
}
```

Example 8: Enumerated Values in JSON Schema

- The enum keyword constrains a field's value to a fixed set of unique values, specified in an Array.
- The Schema in this example limits the set of allowable values in the tags Array to one of "Open Source", "Java", "JavaScript", "JSON", or "REST".

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "tags": {
      "type": "array",
      "minItems": 2,
      "maxItems": 4,
      "items": {
        "enum": ["Open Source", "Java", "JavaScript", "JSON", "REST"]
      }
    }
  },
  "additionalProperties": false,
  "required": ["tags"]
}
```

Example 9: Object in JSON Schema

- JSON Schema enables you to specify an object.
- This is the heart of semantic validation because it enables you to validate Objects exchanged between applications.
- With this capability, both an API's Consumer and Producer can agree on the structure and content of important business concepts such as a person or order.
- The Schema in in this example specifies the content of a speaker Object.

```json
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "speaker": {
      "type": "object",
      "properties": {
        "firstName": { "type": "string" },
        "lastName": { "type": "string" },
        "email": { "type": "string" },
        "postedSlides": { "type": "boolean" },
        "rating": { "type": "number" },
        "tags": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "additionalProperties": false,
      "required": [ "firstName", "lastName", "email",
        "postedSlides", "rating", "tags" ]
    }
  },
  "additionalProperties": false,
  "required": ["speaker"]
}
```

- This Schema is similar to previous examples, with the addition of a top-level speaker object nested inside the root object.

Pattern Properties in JSON Schema

- JSON Schema provides the ability to specify repeating fields (with similar names) through pattern properties (with the **patternProperties** keyword) based on Regular Expressions.
- In this example, the `^line[1-3]$` Regular Expression allows for the following address fields in a corresponding JSON document: `line1`, `line2`, and `line3`.

```
{
  "$schema": "URI/draft-08/schema#",
  "type": "object",
  "properties": {
    "city": {
      "type": "string"
    },
    "state": {
      "type": "string"
    },
    "zip": {
      "type": "string"
    },
    "country": {
      "type": "string"
    }
  },
  "patternProperties": {
    "^line[1-3]$": {
      "type": "string"
    }
  },
  "additionalProperties": false,
```

- Here's how to interpret this Regular Expression:
  - *^ represents the beginning of the string.*
  - *`line` translates to the literal string "line".*
  - *`[1-3]` allows for a single integer between `1` and `3`.*
  - *`$` indicates the end of the string.*
- Note that only `line1` is required, and the others are optional.

```
  "required": ["city", "state",
      "zip", "country", "line1"]
}
```

Regular Expressions in JSON Schema

- JSON Schema also uses Regular Expressions to constrain field values.
- In this example, the Regular Expression specifies a valid email address.
- Here's how to interpret this Regular

```
{
  "$schema": "URI/draft-08/schema#",
  "type": "object",
  "properties": {
    "email": {
    "type": "string",
      "pattern":
      "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
    "firstName": { "type": "string" },
    "lastName": { "type": "string" }
```

Expression:

- *^ represents the beginning of the string.*
- *`[\\w|-|.]+` matches one-to-many instances of the following pattern:*
  - `-[\\w|-|.]` matches a word character (`a-zA-Z0-9_`), a dash (-), or a dot(.).
- *@ indicates the literal "@".*
- *`[\\w]+` matches one-to-many instances of the following pattern:*
  - `[\\w]` matches a word character (a-zA-Z0-9_).
- *`\\.` indicates the literal "."*
- *`[A-Za-z]{2,4}` matches two to four occurrences of the following pattern:*
  - `[A-Za-z]` matches an alphabetic character.
- *$ indicates the end of the string.*

```
    },
    "additionalProperties": false,
    "required": ["email", "firstName", "lastName"]
  }
```

- The double backslash (\\) is used by JSON Schema to denote special characters within regular expressions because the single backslash (\) normally used in standard Regular Expressions won't work in this context.
- This is due to that fact the a single backslash is already used in core JSON document syntax to escape special characters (e.g., \b for a backspace).

Dependent Properties in JSON Schema

- Dependent Properties introduce dependencies between fields in a Schema: *one field depends on the presence of the other.*
- The dependencies keyword is an object that specifies the dependent relationship(s), where field x maps to an array of fields that must be present if y is populated.
- In the example on the right, **tags** must be present if **favoriteTopic** is provided in the corresponding JSON document (that is, `favoriteTopic` depends on `tags`).
- The JSON document in the following example is valid because the

```
{
  "$schema": "URI/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string",
      "pattern":
        "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "tags": {
      "type": "array",
      "items": {
      "type": "string"
      }
    },
    "favoriteTopic": {
      "type": "string"
    }
  },
  "additionalProperties": false,
  "required": ["email", "firstName", "lastName"],
  "dependencies": {
    "favoriteTopic": ["tags"]
  }
}
```

`favoriteTopic` is present, and the tags Array is populated.

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "tags": [
  "JavaScript", "AngularJS", "Yeoman"
  ],
  "favoriteTopic": "JavaScript"
}
```

Internal References in JSON Schema

- **References** provide the ability to reuse definitions/validation rules.
- Think of references as *DRY (Do Not Repeat Yourself)* for JSON Schema - a form of code re-usability.
- References can be either **Internal** (inside the same Schema) or **External** (in a separate/external Schema).

```
{
    "$schema": "URI/draft-04/schema#",
    "type": "object",
    "properties": {
        "email": {
            "$ref": "#/definitions/emailPattern"
        },
        "firstName": {
            "type": "string"
        },
        "lastName": {
            "type": "string"
        }
    },
    "additionalProperties": false,
    "required": ["email", "firstName", "lastName"],
    "definitions": {
        "emailPattern": {
            "type": "string",
```

- In the example on the right, you'll notice that the *Regular Expression* for the email field has been replaced by a **$ref**, a *Uniform Resource Identifier* (URI) to the actual *definition/ validation rule* for the email field:
  - *# indicates that the definition exists locally within the Schema.*
  - */definitions/ is the path to the definitions object in this Schema. Note that the definitions keyword indicates the use of a reference.*
  - *emailPattern is the path to the emailPattern specification within the definitions object.*
  - *JSON Schema leverages JSON Pointer to specify URIs, e.g., #/definitions/emailPattern.*
- We've just moved the definition for email addresses to a *common location*

```
                "pattern":
                  "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
            }
        }
    }
```

that can be **re-used** throughout the Schema by multiple fields.

External References in JSON Schema

- External References in JSON Schema provide a way to specify validation rules in an external Schema file.
- In this case, **Schema A** references **Schema B** for a particular set of *validation rules*.
- External References enable a development team (or several teams) to **re-use** common

- **speaker** Schema that now references an external (second) Schema.

```
{
    "$schema":
      "URI/draft-08/schema#",
    "type": "object",
    "properties": {
        "email": {
            "$ref":
            "schema-URI#/definitions/emailPattern"
        },
        "firstName": {
            "type": "string"
        },
        "lastName": {
            "type": "string"
        }
    },
    "additionalProperties": false,
```

Schemas and definitions across the enterprise.

- The example to the right shows the **speaker** Schema that now *references* an external (second) Schema.
- Notice the two key differences:
  - *The definitions Object has been factored out of this schema. Don't worry; it comes back really soon.*
  - *The email field's $ref now points to an external Schema (ex-14-my-commonschema.json) to find*

```
    "required":
      ["email", "firstName", "lastName"]
}
```

the definition/validation rule for this field. We'll cover the HTTP address to the external Schema later in this chapter.

The External Schema in JSON Schema

- The following is the External Schema that is being referenced in the previous example.
- The definitions object that contains the emailPattern validation rule now resides in the external Schema.

```
{
```

Here's how the two JSON Schemas connect:

- In the referencing schema, the URI prefix before the # in the `$ref` tells the JSON Schema processor to look for the `emailPattern` definition in an external Schema.

```
    "$schema": "schema-version-URI",
    "id": "schema-URL",
    "definitions": {
      "emailPattern": {
        "type": "string",
        "pattern":
          "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
      }
    }
}
```

- In the referenced schema (the external Schema), the `id` field (a JSON Schema keyword) at the root of the Schema makes the content of the Schema available to external access.
- The URI in `$ref` and `id` should be an exact match to make the reference work properly.
- The definitions object works the same as it did for internal references.

End of Topic

What you have learned

- In this topic, you were introduced to the JSON Schema and how it helps in application architecture.
- You have now learnt how to structure and validate JSON instance documents with a JSON Schema.

Further study resources

- In addition to the json-schema.org site mentioned previously, here are a few more resources;
- Using JSON Schema by Joe McIntyre provides a wealth of JSON Schema-related reference information and tools, including these:
  - *The Using JSON Schema*
  - *The jsonvalidate application*
  - *The ujs-validate npm module*
- Understanding JSON Schema by Michael Droettboom et al.

- A Short Guide to JSON Schema