# JSON AND THRIFT

## INTERNET ENGINEERING

Fall 2022

@1995parham

- Introduction
- Documentation & Validation
- Processing (using JavaScript)
- Thrift
- Conclusion

- Introduction
- Documentation & Validation
- Processing (using JavaScript)
- Thrift
- Conclusion

# INTRODUCTION

- HTML + CSS + JavaScript: Interactive Web pages
    - Web server is not involved after page is loaded
    - JavaScript reacts to user events
- However, most web applications needs data from server after the page is loaded
    - A common (standard) format to exchange data
    - A mechanism to communication: `fetch`
- (Almost) Always the data is structed

# INTRODUCTION (CONTD.)

- In general (not only in web) to **store** or **transport** data, we need a common format, to specify the structure of data; e.g.,
    - Documents: PDF, DOCx, PPTx, ...
    - Objects: Java Object Serialization/Deserialization

- How to define the data structure?
    - *Binary* format (similar to binary files)
        - Difficult to develop & debug
        - machine depended
        - ...
    - *Text* format (similar to text files)
        - Easy to develop & debug
        - human readable
        - ...

# INTRODUCTION (CONT.)

- Example: Data structure of a class
    - Course name, teacher, # of students, each student information

```go
type Course struct {
  Name     string
  Teacher  string
  Students []Student
  Capacity int
}

type Student struct {
  FirstName string
  LastName  string
  ID        string
}

c := Course {
  Name: "IE",
  Teacher: "Bahador Bakhshi",
  Students: []Student{
    { FirstName: "Parham", LastName: "Alvani", ID: "9231058" },
```

```json
{
  "name": "IE",
  "teacher": "Bahador Bakhshi",
  "students": [
    { "first_name": "Parham", "last_name": "Alvani", "id": "9231058" }
  ],
  "capacity": 30
}
```

```
A4                                     # map(4)
   64                                  # text(4)
      6E616D65                         # "name"
   62                                  # text(2)
      4945                             # "IE"
   67                                  # text(7)
      74656163686572                   # "teacher"
   6F                                  # text(15)
      42616861646F722042616B68736869 # "Bahador Bakhshi"
   68                                  # text(8)
      73747564656E7473                 # "students"
   81                                  # array(1)
      A3                               # map(3)
         6A                            # text(10)
            66697273745F6E616D65     # "first_name"
         66                            # text(6)
            50617268616D               # "Parham"
         69                            # text(9)
```

```
IE
Bahador Bakhshi
30
1
Parham
Alvani
9231058
```

# JSON

- JavaScripters’ approach
- JSON: JavaScript Object Notation
- Data is represented as a JS (POD) object
- Standards: RFC 8259, ECMA-404

```
{
  "name": "IE",
  "teacher": "Bahador Bakhshi",
  "students": [
    { "first_name": "Parham", "last_name": "Alvani", "id": "9231058" }
  ],
  "capacity": 30
}
```

# JSON SYNTAX

- Data is in name-value pairs
  - Field name in double quotes, followed by a colon, followed by a value
  - In JSON, the *keys* **must** be strings
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

- Data Types:
  - string

```
"This is a string"
```

  - number

```
42
3.1415926
```

  - object

```
{ "key1": "value1", "key2": "value2
```

  - array

```
[ "first", "second", "third" ]
```

  - boolean

```
true
false
```

# WHY TO STUDY JSON: BENEFITS

- Simplify data sharing & transport
    - JSON is text based and platform independent
- JSON is simple, efficient, and popular
- Extensive libraries to process JSON
    - To validate, to present, …
- In web application, data separation from HTML
    - E.g., table structure by HTML, table data by JSON

- Introduction
- Documentation & Validation
- Processing (using JavaScript)
- Thrift
- Conclusion

# DOCUMENTATION & VALIDATION

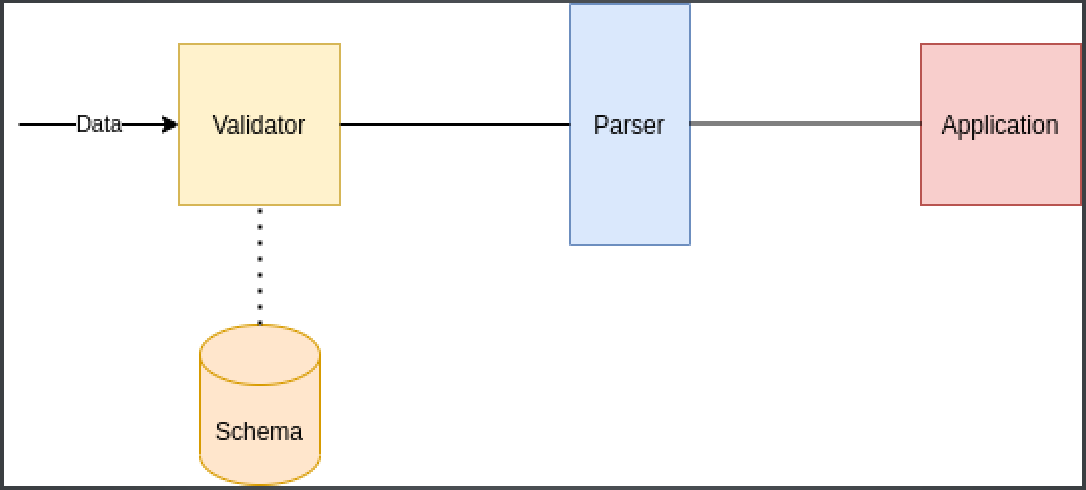
- Assume that application A exchange data with application B
- How does A's developer document the data format?
    - How does the receiver know the structure of the data?
        - In English?
        - By samples?
- How can the receiver validate the data?

# VALID DATA

- Syntax
    - Syntax rules
        - E.g., all keys must be double quoted in JSON
    - Error makes the parser fails to parse the file
- Symantec (structure)
    - Application specific rules
        - E.g. student must have ID
    - Error makes the application fails

# HOW TO VALIDATE STRUCTURE?

- Application specific programs need to check structure of data
    - Different applications needs different programs
    - Change in data structure needs code modification
- General validator + reference document
    - Reference document
    - JSON: JSON Schema


Data
Validator
Parser
Application
Schema

# THE REFERENCE DOCUMENTS USAGE

- The Reference document is the answer of
  - Documentation: It describes the structure of data which is human readable
  - Interaction: The description is machine readable
  - Validation: There are validators to validate the data based on it

# JSON SCHEMA

- JSON schema is JSON also
- The JSON document being validated or described we call the instance, and the document containing the description is called the schema.

# HELLO WORLD

- This accepts anything, as long as it’s valid JSON

```
{}
```

- The most common thing to do in a JSON Schema is to restrict to a specific type. The `type` keyword is used for that.

```
{ "type": "string" }
```

# DECLARING A JSON SCHEMA

- Since JSON Schema is itself JSON, it’s not always easy to tell when something is JSON Schema or just an arbitrary chunk of JSON.
- The `$schema` keyword is used to declare that something is JSON Schema.
- It’s generally good practice to include it, though it is not required.

```
{ "$schema": "http://json-schema.org/draft-07/schema#" }
{ "$schema": "http://json-schema.org/draft/2019-09/schema#" }
```

# DECLARING A UNIQUE IDENTIFIER

- It is also best practice to include an `$id` property as a unique identifier for each schema.
- For now, just set it to a URL at a domain you control, for example:

```
{ "$id": "http://yourdomain.com/schemas/myschema.json" }
```

# ANNOTATIONS

- JSON Schema includes a few keywords, `title`, `description`, `default`, `examples` that **aren’t strictly used for validation**, but are used to describe parts of a schema.
- The `title` and `description` keywords must be strings.
- A “title” will preferably be short, whereas a “description” will provide a more lengthy explanation about the purpose of the data described by the schema.
- The `default` keyword specifies a default value for an item.

# STRING

```
{ "type": "string" }
```

- Length
  - The length of a string can be constrained using the `minLength` and `maxLength` keywords.
  - For both keywords, the value must be a non-negative number.
- Regular Expressions
  - The `pattern` keyword is used to restrict a string to a particular regular expression.

# STRING (CONTD.)

- Format
    - The `format` keyword allows for basic semantic validation on certain kinds of string values that are commonly used.
        - Dates and times
        - Email addresses
        - Hostnames
        - IP Addresses
        - ...

# NUMERIC TYPES

- The integer type is used for integral numbers.

```
{ "type": "integer" }
```

- The number type is used for any numeric type, either integers or floating point numbers.

```
{ "type": "number" }
```

# NUMERIC TYPES (CONTD.)

- Multiples
    - Numbers can be restricted to a multiple of a given number, using the `multipleOf`
    - It may be set to any positive number.
- Range
    - Ranges of numbers are specified using a combination of the `minimum` and `maximum` keywords

# OBJECT

- Objects are the mapping type in JSON.

```
{ "type": "object" }
```

- Properties
  - The properties (key-value pairs) on an object are defined using the `properties` keyword.
  - The value of `properties` is an object, where each key is the name of a property and each value is a JSON schema used to validate that property.

# OBJECT (CONTD.)

```
{
  "type": "object",
  "properties": {
    "number":      { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "type": "string",
                     "enum": ["Street", "Avenue", "Boulevard"]
                   }
  }
}
```

# OBJECT (CONTD.)

- The `additionalProperties` keyword is used to control the handling of extra stuff
- properties whose names are not listed in the `properties` keyword.
- By default any additional properties are **allowed**
- If `additionalProperties` is an object, that object is a schema that will be used to validate any additional properties not listed in properties.

```json
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "enum": ["Street", "Avenue", "Boulevard"] }
  },
  "additionalProperties": { "type": "string" }
}
```

```json
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "enum": ["Street", "Avenue", "Boulevard"] }
  },
  "additionalProperties": false
}
```

# OBJECT (CONTD.)

- By default, the properties defined by the properties keyword are *not required*.
- The `required` keyword takes an array of zero or more strings.

# OBJECT (CONTD.)

```
{
  "type": "object",
  "properties": {
    "name":      { "type": "string" },
    "email":     { "type": "string" },
    "address":   { "type": "string" },
    "telephone": { "type": "string" }
  },
  "required": ["name", "email"]
}
```

# ARRAY

- Arrays are used for ordered elements.
- In JSON, each element in an array may be of a different type.

```
{ "type": "array" }
```

- List validation is useful for arrays of arbitrary length where each item matches the *same schema*.
- For this kind of array, set the `items` keyword to a single schema that will be used to validate all of the items in the array.

# ARRAY (CONTD.)

```
{
  "type": "array",
  "items": {
    "type": "number"
  }
}
```

# EXAMPLE

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Product",
  "type": "object",
  "properties": {
    "id": {
      "type": "number",
      "description": "Product identifier"
    },
    "name": { "type": "string" },
    "price": { "type": "number", "minimum": 0 },
    "tags": {
      "type": "array",
      "items": { "type": "string" }
    },
    "stock": {
      "type": "object",
      "properties": {
```

# EXAMPLE (CONTD.)

```
{
  "id": 1,
  "name": "Foo",
  "price": 123,
   "tags": [
    "Bar",
    "Eek"
  ],
  "stock": {
   "warehouse": 300,
   "retail": 20
  }
}
```

# JSON SCHEMA VALIDATOR

- Validators available
    - As Online tools
    - As programming languages libraries
    - Standalone tools

- Introduction
- Documentation & Validation
- Processing (using JavaScript)
- Thrift
- Conclusion

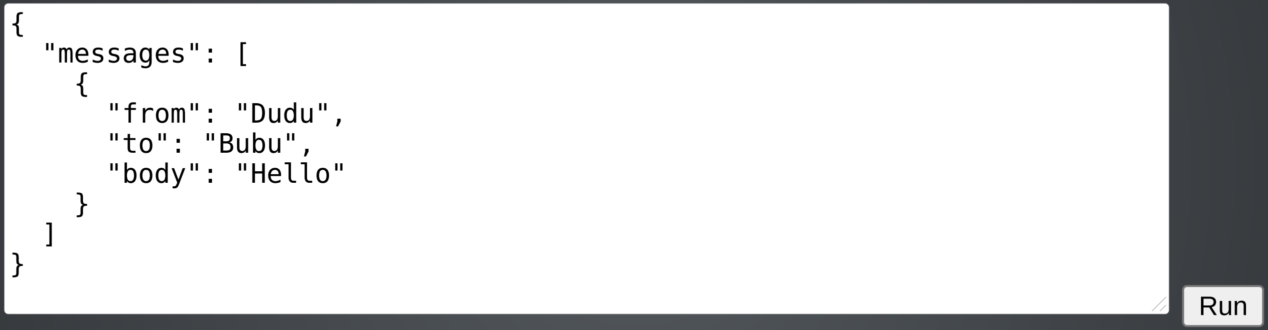
- `The JSON object`, has two very useful methods to deal with JSON-formatted content
    - `JSON.parse()` takes a JSON string and transforms it into a JavaScript object
    - `JSON.stringify()` takes a JavaScript object and transforms it into a JSON string

```
let myObj = { a: '1', b: 2, c: '3' };
let myObjStr = JSON.stringify(myObj);
console.log(myObjStr);
console.log(JSON.parse(myObjStr));
```

Run

# EXAMPLE MESSAGE PARSER

```
{
  "type": "object",
  "properties": {
    "messages": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "from": {
            "type": "string"
          },
          "to": {
            "type": "string"
          },
          "body": {
            "type": "string"
          }
        }
```



```
{
  "messages": [
    {
      "from": "Dudu",
      "to": "Bubu",
      "body": "Hello"
    }
  ]
}
```

```
function parseJSON() {
  output = "";
  input = document.getElementById("json-in-2").value;
  jsonData = JSON.parse(input);

  for (i = 0; i < jsonData.messages.length; i++) {
    msg = jsonData.messages[i];
    output += `
        ${msg.from} sent the following message to ${msg.to}<br />${msg.body}<hr />
  }
  document.getElementById("json-out-2").innerHTML = output;
}
document.getElementById("json-btn-2").onclick = parseJSON;
```

- Introduction
- Documentation & Validation
- Processing (using JavaScript)
- Thrift
- Conclusion

# BASED ON

- Alireza Mohammadi
- Amir Hallaji Bidgoli
- Spring 2021

# INTRODUCTION

- Apache Thrift is an open source, cross-language serialization and remote procedure call (RPC) framework.
- With support for more than 20 programming languages, Apache Thrift can play an important role in many distributed application solutions.

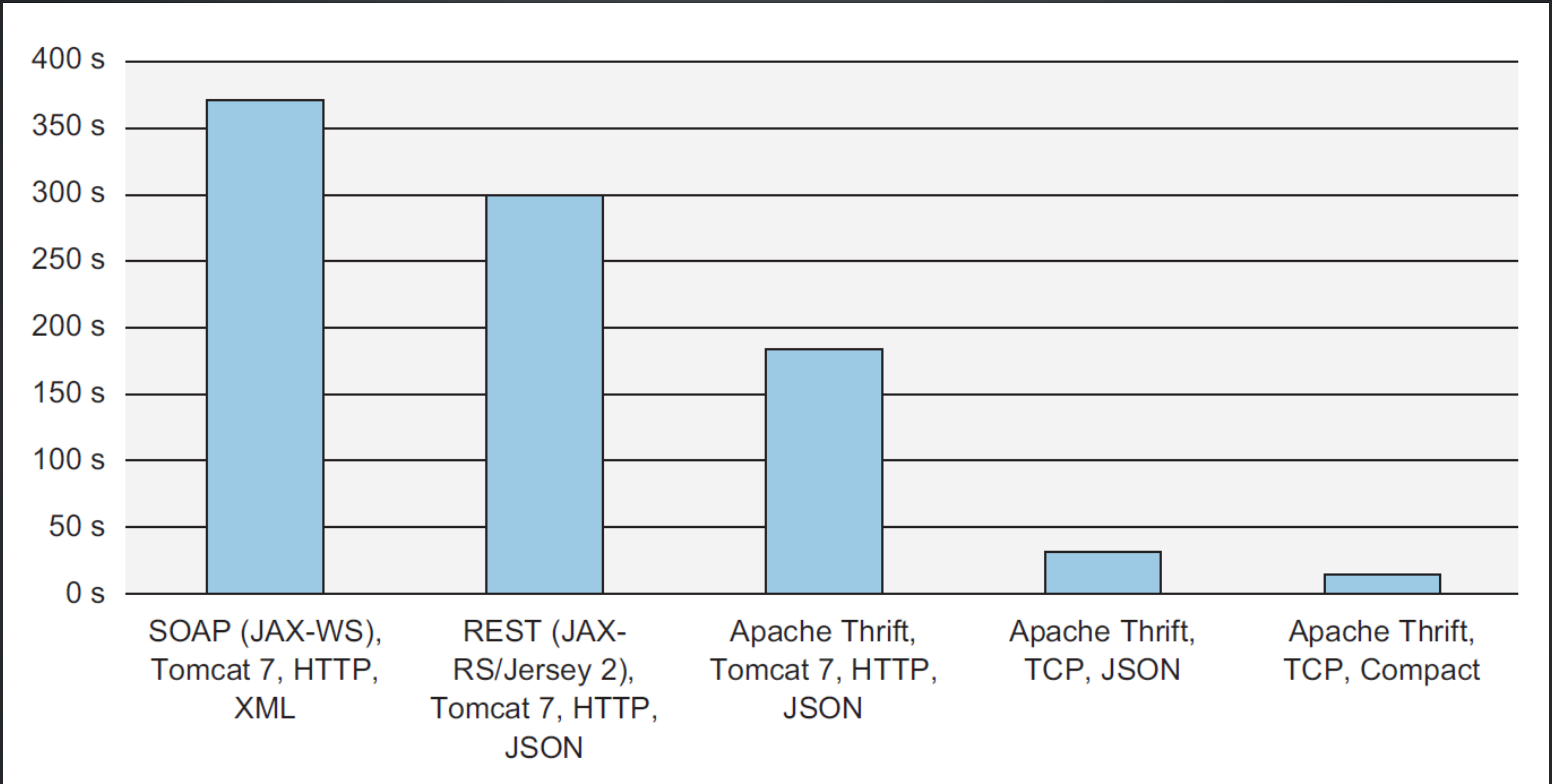
# INTRODUCTION (CONT.)

- As a serialization platform, it enables efficient cross-language storage and retrieval of a wide range of data structures.
- As an RPC framework, Apache Thrift enables rapid development of complete cross-language services with little more than a few lines of code.

# INTRODUCTION (CONT.)

Languages supported by Apache Thrift :

| | | |
|---|---|---|
| Go | C | Python |
| JavaScript | C++ | TypeScript |
| OCaml | Objective-C | ActionScript |
| Ruby | Haxe | Cappuccino |
| AS3 | Node.js | Cocoa |
| D | Php | Elixir |
| Dart | Smalltalk | Scala |
| Haskell | C# | Swift |
| Lua | Erlang | Delphi |
| Perl | Java | Rust |


400 s
350 s
300 s
250 s
200 s
150 s
100 s
50 s
0 s
SOAP (JAX-WS), Tomcat 7, HTTP, XML
REST (JAX-RS/Jersey 2), Tomcat 7, HTTP, JSON
Apache Thrift, Tomcat 7, HTTP, JSON
Apache Thrift, TCP, JSON
Apache Thrift, TCP, Compact

# BRIEF HISTORY

- It was developed at Facebook and it is now an open source project in the Apache Software Foundation.
- The implementation was described in an April 2007 technical paper released by Facebook, now hosted on Apache.

# THRIFT DEFINITION FILE

```
/**
 * Thrift files can reference other Thrift files to include common struct
 * and service definitions. These are found using the current path, or by
 * searching relative to any paths specified with the -I compiler flag.
 *
 * Included objects are accessed using the name of the .thrift file as a
 * prefix. i.e. shared.SharedObject
 */
include "shared.thrift"

/**
 * You can define enums, which are just 32 bit integers. Values are optional
 * and start at 1 if not supplied, C style again.
 */
enum Operation {
 ADD = 1,
 SUBTRACT = 2,
 MULTIPLY = 3,
 DIVIDE = 4,
```

# PYTHON CLIENT

```python
if __name__ == "__main__":
    # Make socket
    transport = TSocket.TSocket('localhost', 9090)

    # Buffering is critical. Raw sockets are very slow
    transport = TTransport.TBufferedTransport(transport)

    # Wrap in a protocol
    protocol = TBinaryProtocol.TBinaryProtocol(transport)

    # Create a client to use the protocol encoder
    client = Calculator.Client(protocol)

    # Connect!
    transport.open()

    client.ping()
    print('ping()')
```

# JAVA SERVER

```
public static void main(String [] args) {
   try {
     handler = new CalculatorHandler();
     processor = new Calculator.Processor(handler);

     Runnable simple = new Runnable() {
       public void run() {
         simple(processor);
       }
     };

     new Thread(simple).start();
     new Thread(secure).start();
   } catch (Exception x) {
     x.printStackTrace();
   }
}
```

# JAVA HANDLER

```java
public class CalculatorHandler implements Calculator.Iface {

  private HashMap<Integer,SharedStruct> log;

  public CalculatorHandler() {
    log = new HashMap<Integer, SharedStruct>();
  }

  public void ping() {
    System.out.println("ping()");
  }

  public int add(int n1, int n2) {
    System.out.println("add(" + n1 + "," + n2 + ")");
    return n1 + n2;
  }

  public int calculate(int logid, Work work) throws InvalidOperation {
```

- Introduction
- Documentation & Validation
- Processing (using JavaScript)
- Thrift
- Conclusion

# WHAT ARE THE NEXT?!

- Other related technologies in data exchange
    - Protocol Buffers
    - Thrift
    - YAML (YAML Ain't Markup Language)

# REFERENCES 📚

- https://json-schema.org/
- Prof. Bahador Bakhshi's Internet Eng. Course's Slides