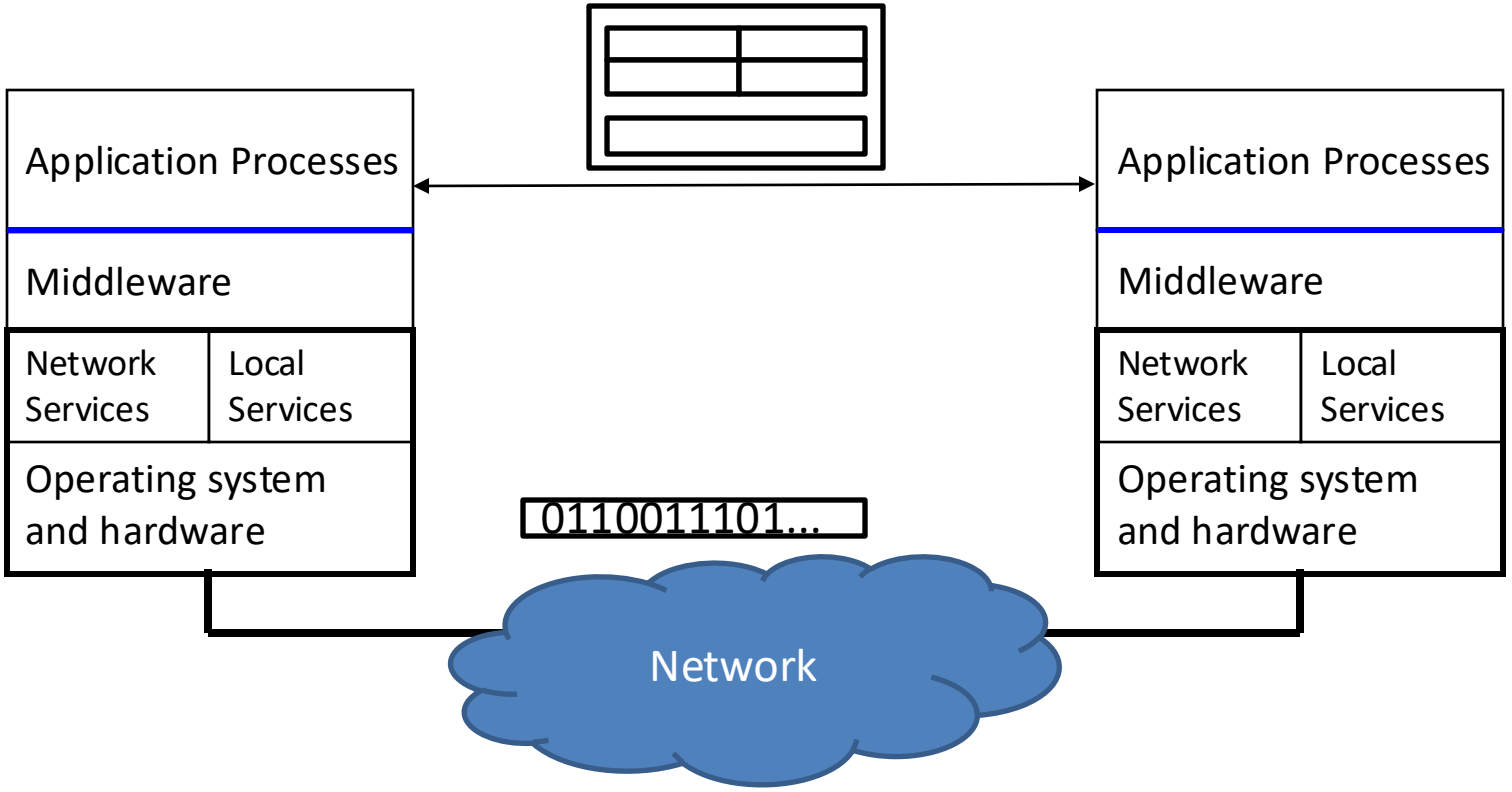
Abstract Syntaxes and Schemas

Reference for study: Michael Droettboom,
"Understanding JSON Schema"

https://json-schema.org/understanding-json-schema/

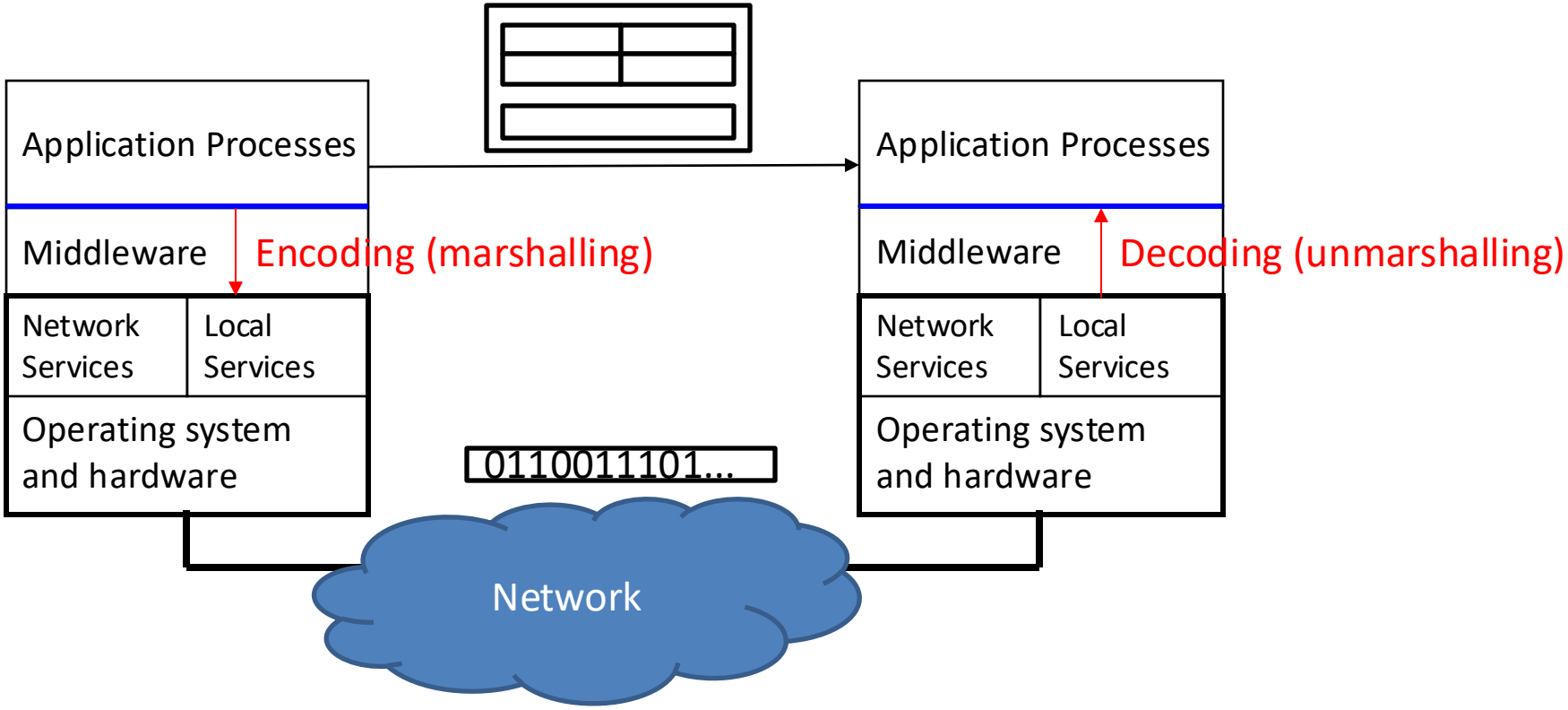
Data Transparency

- Protocols must ensure that data are correctly transferred despite systems are heterogeneous

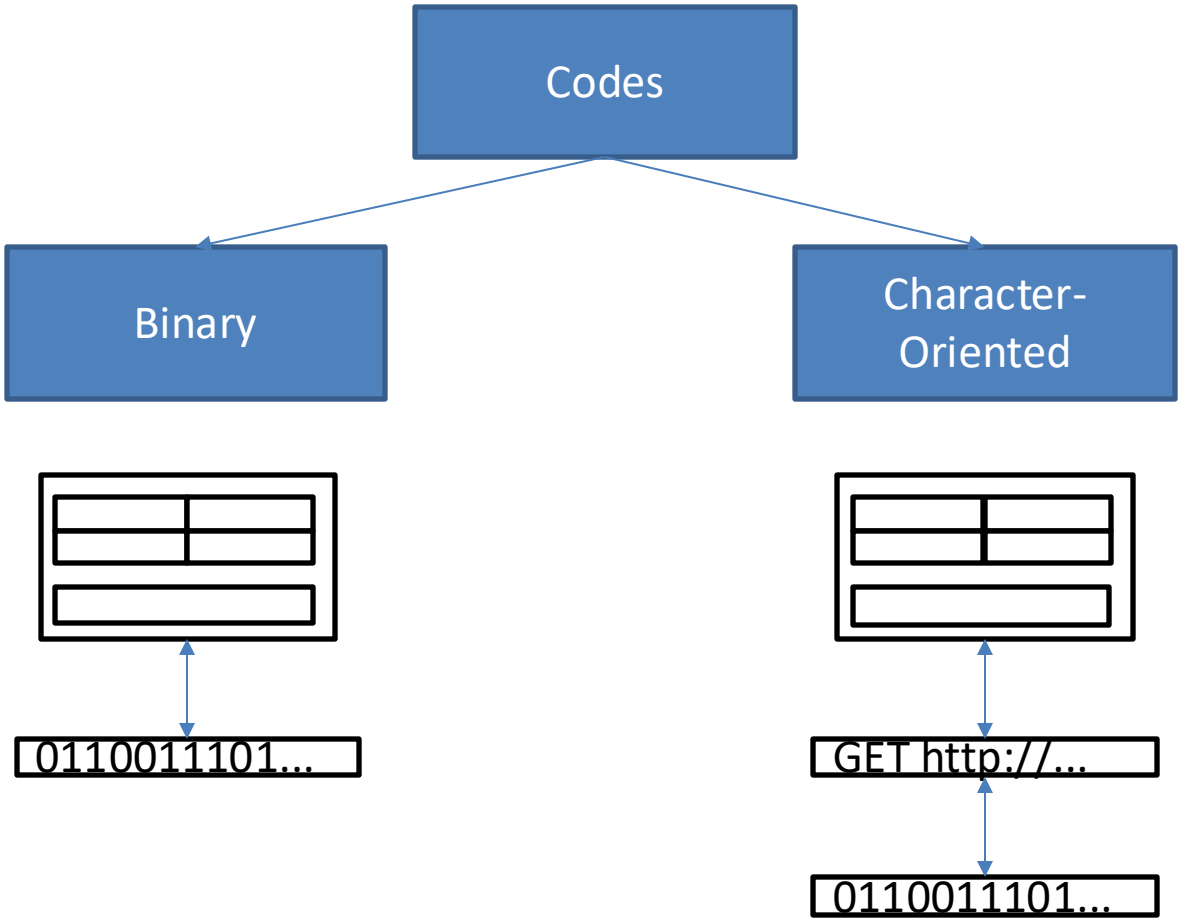


Data Transparency

- Data Transparency is obtained by properly encoding and decoding data, according to a protocol



Different Types of Codes



Different Types of Codes

- Codes are defined so that the receiver can **separate**, **validate** and **decode** messages
- The HTTP solution (character oriented)
  - Different codes can be used
  - Initially the code is plain ASCII
  - The code used in the body is communicated / negotiated in the header

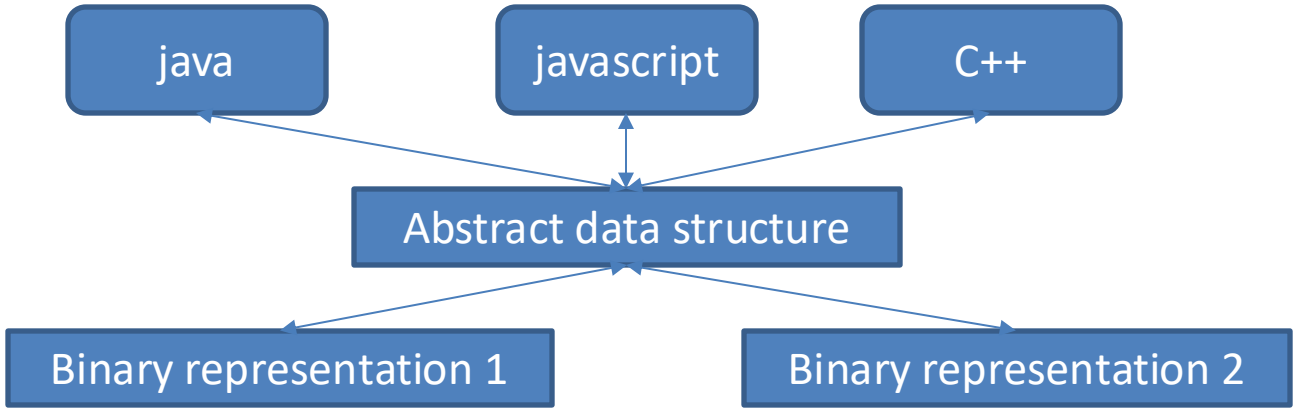
```
Content-type: application/json; charset=utf-8
Content-encoding: gzip
Content-length: 22456
```

Applications

- Applications typically exploit standard ways for system-independent data representation
  - **Binary solutions**: ASN.1, XDR, Protocol Buffers,...
  - **Character-oriented solutions**: XML, JSON, YAML,...
- These standards typically include:
  1. A language to define ***abstract* data types (abstract syntax)**
  2. "Neutral" (system independent) data representations for any abstract data type that can be defined by the language
     - include a mechanism to correctly separate/validate/decode data
     - may require the receiver to know the data type in advance or to learn the type from the data

Why Abstract Syntaxes

- Language/system independency
  - Programmers can use any programming language
  - Libraries translate automatically to/from the language data types
- Independency from binary encoding (different binary codes can be used for the same abstract data)
  - Example: ASN.1 admits different binary codes (BER, DER)



Why Abstract Syntaxes

- Abstract syntaxes can be used to **specify/validate** message syntax
  - Service users have unambiguous specifications of interface data
  - The programmer can delegate validation to standard validators
    - Better security and reliability/robustness of applications
    - More simplicity of application code

```
app.post('/user', [
  check('username').isEmail(), // username must be an email
  check('password').isLength({ min: 5 }) // password must be at least 5 chars long
] , (req, res) => {
const errors= validationResult(req);
if (!errors.isEmpty()) {
return res.status(422).json({ errors: errors.array() });
}
```

Validation solution based on express-validator

No formal spec./doc.

Only simple checks possible

Validation code "pollutes" app

Abstract Syntaxes for Character-Oriented Representations

- XML
  - W3C XML Schema (well established standard)
- JSON
  - JSON Schema (more recent, still IETF draft)

- ...
- Powerful but not panacea
  - expressiveness has limits: still need to validate something in the application

The JSON Schema Language

- Based on JSON itself
  - A schema is a JSON object with special meaning
  - Its format is described by a schema (auto-description)
    - `http://json-schema.org/draft-07/schema`
- IETF Draft
  - Current version 2020-12, but we refer to **draft-07**
- A schema defines a data type, i.e. a set of valid JSON values (and corresponding strings)
  - JSON data types correspond to types commonly available in any programming language:
    - Primitive data (string, boolean, number, null), Arrays, Objects

Machine readable!

serializable

Can be validated

Example

The schema this JSON object refers to

Schema language

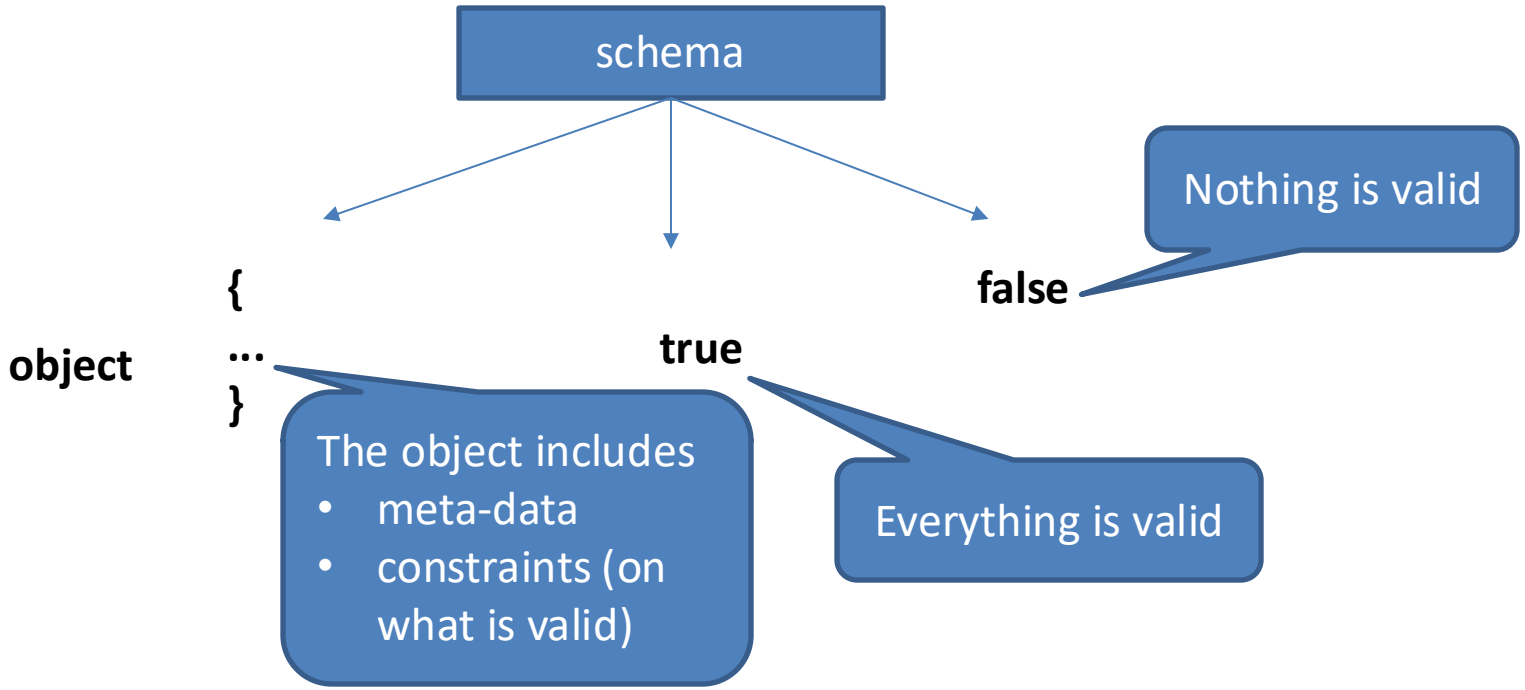
Schema id

Meta-data

Type description (constraints)

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "http://dsp.polito.it/test1schema.json",
    "type": "object",
    "properties": {
        "firstname": { "type": "string" },
        "lastname": { "type": "string" },
        "birthdate": { "type": "string", "format": "date" },
        "address": {
            "type": "object",
            "properties": {
                "street": { "type": "string" },
                "city": { "type": "string" },
                "country": { "type": "string" }
            }
        }
    }
}
```

Schema Structure



Other Meta-Data

- Annotations
  - title
  - description
  - default
  - examples
- Comments
  - $comment

The Constraints

- Type (**type** property)
  - defines the type(s) of the data structure:
    - **string** Unicode string
    - **number** Real number
    - **integer** Integer number
    - **object** Collection of properties (named typed values)
    - **array** Ordered list of typed values (different values may have different types)
    - **boolean** Boolean value (true/false)
    - **null** Null value

Can be an array of allowed types

- Type-dependent constraints (additional properties)

Types: string

- Additional constraints:
  - **minLength**
  - **maxLength**
  - **format** (date, time, date-time, email, hostname, ipv4, ipv6, iri, uri, uri-template, uri-reference, iri-reference, regex)
  - **pattern** (regular expression)

Types: number and integer

- Additional constraints:
  - multipleOf
  - minimum
  - exclusiveMinimum
  - maximum
  - exclusiveMaximum

No automatic conversion from/to strings!

"type": "number"	123 ✓	"123" ✗
" type": "string"	123 ✗	"123" ✓

Types: object

- Additional constraints (about properties):
  - **properties** (constraints about some object properties)
    - the value is an object
    - The object properties are the constraints:

      key: name of constrained property

      value: schema specifying type of constrained property value

```
{
    "type": "object",
    "properties": {
        "firstname": { "type": "string" },
        "lastname": { "type": "string" },
        ...
    }
}
```

Name of constrained property

Schema specifying value of constrained property

Types: object

- Additional constraints (about properties):
  - **additionalProperties** (constraints about additional properties)
    - If not specified, any additional property is valid
    - If specified, its value can be:

      true: any additional property is valid (default)

      false: no additional property is valid

      a schema: any additional property must obey to it
  - **required**
    - If not specified, all properties are optional
    - If specified, its value is an array (names of the required properties)
  - **propertyNames**
    - If not specified, all property names are admitted
    - If specified, its value is a **schema** the property names must obey (the schema type is implied to be string)

Types: object

- Additional constraints (about number of properties):
  - **minProperties**
    - Minimum number of properties
  - **maxProperties**
    - Maximum number of properties

Types: object

- Additional constraints (dependencies):
  - **dependencies** (constraints about property dependencies)
    - If specified, its value must be an object whose properties express the dependencies
- Two types of dependencies:
  - **Property dependencies**
    - If a property is present, then also certain other properties must be present
  - **Schema dependencies**
    - If a property is present, the object schema must be extended with certain additional constraints

Types: object

- Dependencies examples

```
{
   "type": "object",
   "properties": {
      "firstname": { "type": "string" },
      "lastname": { "type": "string" },
      "age": { "type": "integer", "minimum": 0 },
      "idnum": { "type": "string" },
      ...
   }
    "dependencies": {
      "firstname": [ "lastname" ],
      "idnum": {"properties": {"age": { "type": "integer",
                                        "minimum": 10 }
                               }
               }
    }
}
```

If there is "firstname", there must be also "lastname"

Names of properties that have dependencies

If there is "idnum", age must be at least 10"

Types: array

- Additional constraints (about number of items):
  - **minItems**
    - Minimum number of items
  - **maxItems**
    - Maximum number of items
- Additional constraints (about uniqueness of items):
  - **uniqueness**
    - If true, items must be unique (no duplicates)

Types: array

- Additional constraints (about items types):
  - **items** (constraints about the array items types)
    - If the value is **a schema** (object): each item must obey the schema
    - If the value is **an array** of schema objects: each item must obey the *corresponding* schema
  - **additionalItems** (constraints about additional array items types)
    - If the value is **true** (default): additional items allowed after tuple
    - If the value is **false**: no additional items allowed after tuple
    - If the value is **a schema** (object) : each additional item must obey it
  - **contains** (constraints about the array items types)
    - If the value is **a schema** (object): *one or more* items must obey the schema
- Example: allow multiple addresses in the sample schema

list

tuple

Types: boolean and null

- No additional constraints

No automatic conversion from/to other types!

"type": "boolean"	true ✓	"true" ✗	1 ✗	
" type": "null"	null ✓	"null" ✗	0 ✗	false ✗

Types defined by Enumeration

- **enum** property
  - Specifies the allowed values by enumeration (using JSON syntax)
- Can be combined with other type constraints
- Example:

```
{
   "type": "object",
   "properties": {
      "firstname": { "type": "string" },
      "lastname": { "type": "string" },
      "sex": { "enum": [ "male", "female" ] },
   }
}
```

Schema Combination

- A schema can be the result of a combination of schemas
  - **allOf** require validity against all sub-schemas
  - **anyOf** require validity against at least one sub schema
  - **oneOf** require validity against exactly one sub-schema
  - **not** require not validity
- Example:

```
{
    "anyOf": [
        { "type": "number", "multipleOf": 3},
        { "type": "number", "multipleOf": 5 }
    ]
}
```

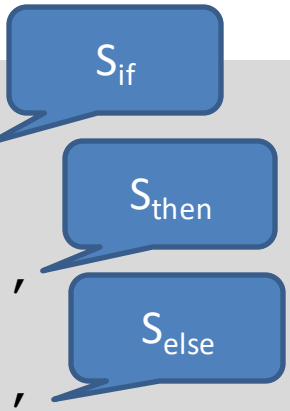
Conditional Schema Combination

- Schemas can be combined by if-then-else constructs
  - If data are valid against S_{if} then data must be valid against S_{then} else data must be valid against S_{else}
  - Example:

```
{
    "if": { "type": "integer", "multipleOf": 3},

    "then": { "type": "number", "multipleOf": 5},

    "else": { "type": "number", "multipleOf": 7},
}
```



Reusing Schemas

- Schemas can be reused by means of references
  - A reference to an existing schema can be used wherever a schema is required
- Example: purchase order

```
{
   "type": "object",
   "properties": {
      "orderDate": { "type": "date" },
      "quantity": { "type": "integer" },
      "shippingAddress": { "$ref": "..." },
      "billingAddress": { "$ref": "..." },
      ...
   }
}
```

URI-like references to internal or external schema

- Reusing internal schema using JSON pointer

```
{
  "definitions": {
    "address": {
      "$id": "#definitions/address",
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" },
        "country": { "type": "string" }
      }
    }
  },
  "type": "object",
    "properties": {
      "orderDate": { "type": "string", "format": "date"},
      "quantity": { "type": "integer" },
      "shippingAddress": { "$ref": "#definitions/address" },
      "billingAddress": { "$ref": "#definitions/address" },
      ...
    }
}
```

- Reusing external schema

```
{
"type": "object",
    "properties": {
      "orderDate": { "type": "string", "format": "date" },
      "quantity": { "type": "integer" },
      "shippingAddress": { "$ref": "addressschema.json" },
      "billingAddress": { "$ref": "addressschema.json" },
      ...
    }
}
```

addressschema.json

```
{
  "$id": "#address",
  "type": "object",
  "properties": {
    "street": { "type": "string" },
    "city": { "type": "string" },
    "country": { "type": "string" }
  }
}
```

Extending Schemas

- Extension (i.e., addition of constraints) can be achieved by combining a referenced schema with another one (e.g., by allOf)
- Exercise: write a schema for the billing address that extends the shipping address schema by including another field (vat number, required)

Schema Validation with Express/ajv

```
var { Validator, ValidationError } = require('express-json-validator-middleware');
var userSchema= JSON.parse(fs.readFileSync(schemaFileName));
var validator = new Validator({ allErrors: true });
validator.ajv.addSchema(userSchema);
var validate = validator.validate;
```

```
app.post('/user', validate({ body: userSchema }) , (req, res) => {
...
}
```

```
app.use(function(err, req, res, next) {
   if (err instanceof ValidationError) {
      res.status(400).send('Invalid request');
      next();
   } else next(err);
});
```

Error handler for validation errors