JSON Schema and Open API

Validating APIs

- Standardize the data going into and coming out of API
  - Better user experience
  - More robust application
  - Better documentation
- RESTful API is very relaxed about request/response
  - Unlike SOAP Web Services, many layers of validation
- Useful to adopt some of the rigor from SOAP Web Services
  - As the implementation of the API grows in size and complexity
- Ensure
  - Data used in requests and responses are valid
  - REST endpoints are receive valid requests and produces expected responses

Standardize Response

- Use a standard response envelope
  - Easier for client to handle the response
- Provides a framework for responses to be returned
- API providers may use response envelope as a way to return the result of the request
  - Eg. Status field in envelope
  - Status code can be application status and/or HTTP status
- If envelope contains HTTP status consider
  - How this affects the client

Response Envelope

- Response envelope may contain
  - Status - the status of the response - may be application and/or HTTP status
  - Data - the data
  - Timestamp, version
  - Pagination information - size of the requested resource, current cursor position, etc.
  - List of available actions
- See OpenWeatherMap example
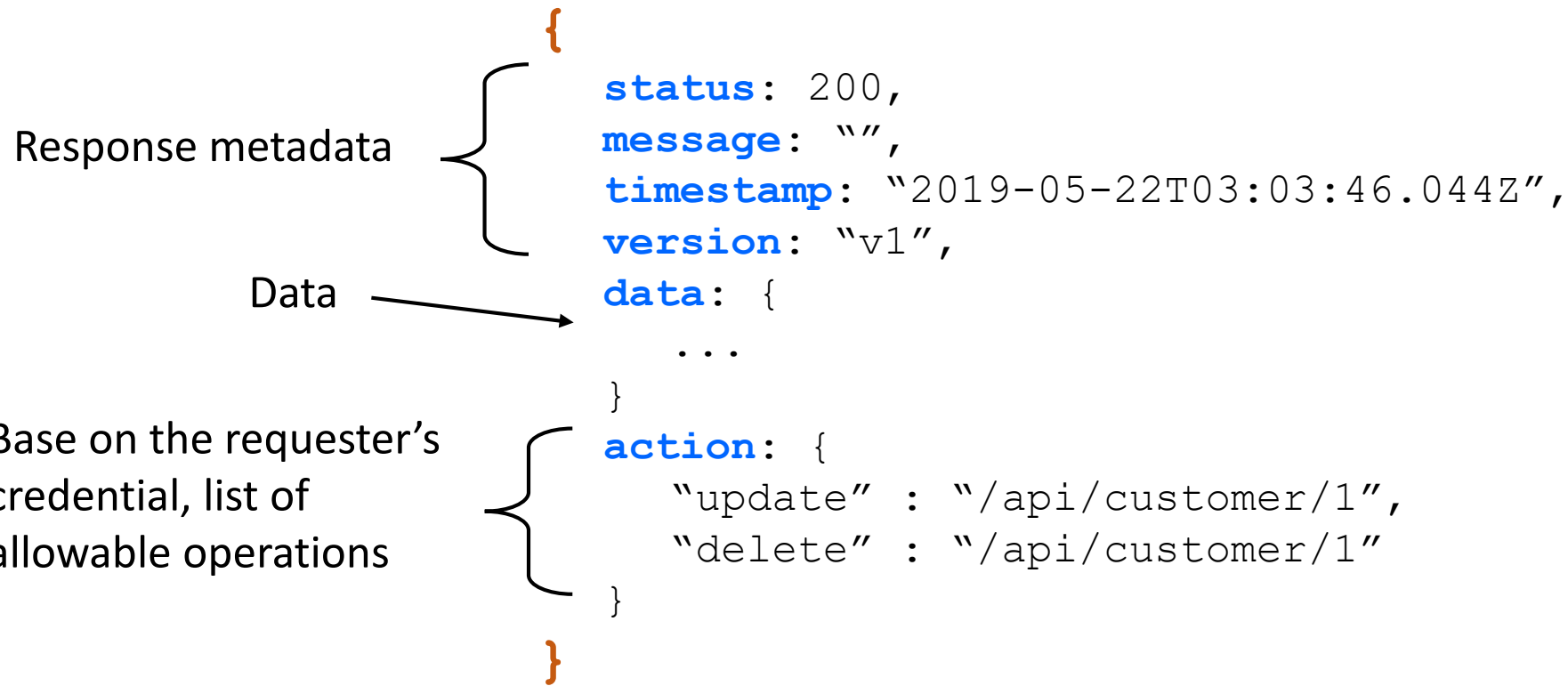 - https://samples.openweathermap.org/data/2.5/forecast?q=M%C3%BCnchen,DE&appid=b6907d289e10d714a6e88b30761fae22

Example Response Envelope

GET /api/customer/1
Accept: application/json

200 OK
Content-Type: application/json

```
{
    status: 200,
    message: "",
    timestamp: "2019-05-22T03:03:46.044Z",
    version: "v1",
    data: {
        ...
    }
    action: {
        "update" : "/api/customer/1",
        "delete" : "/api/customer/1"
    }
}
```

Response metadata

Data

Base on the requester's credential, list of allowable operations

Response envelope, the blue are standard attributes with fixed semantics that clients can rely on

JSON as Data Exchange Format

- Need a common format when exchanging data between platforms and systems
    - Especially if data is richly structured
- JSON is a popular format
    - Simple and natively supported by JavaScript
    - Lots of libraries in other languages to parse, build and process JSON data
- Weaknesses
    - Not very efficient as it is text based
        - Can be encoded with MessagePack - https://msgpack.org
    - No official/standard namespace support - workaround
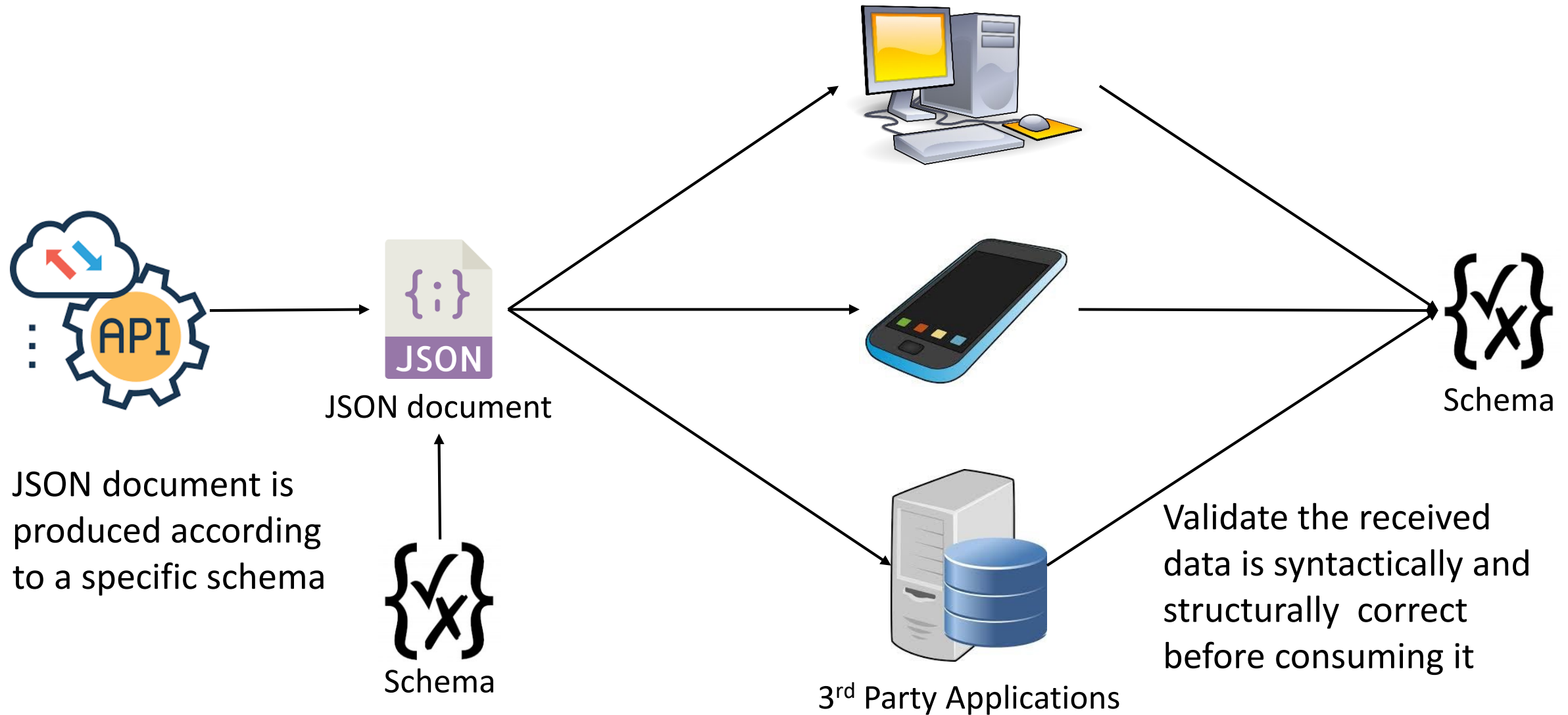   - Initial JSON does not have schema

Schema

- Language agnostic way of describing the structure of data
  - Validate data by comparing with its corresponding schema
  - Serves as a documentation
- Typically JSON are validated by the application
  - To understand the requirement of a piece of JSON, need to read the code that validates it
  - Makes validation across heterogeneous environment more difficult
- Validation
  - Used by the sender to validate before send the data
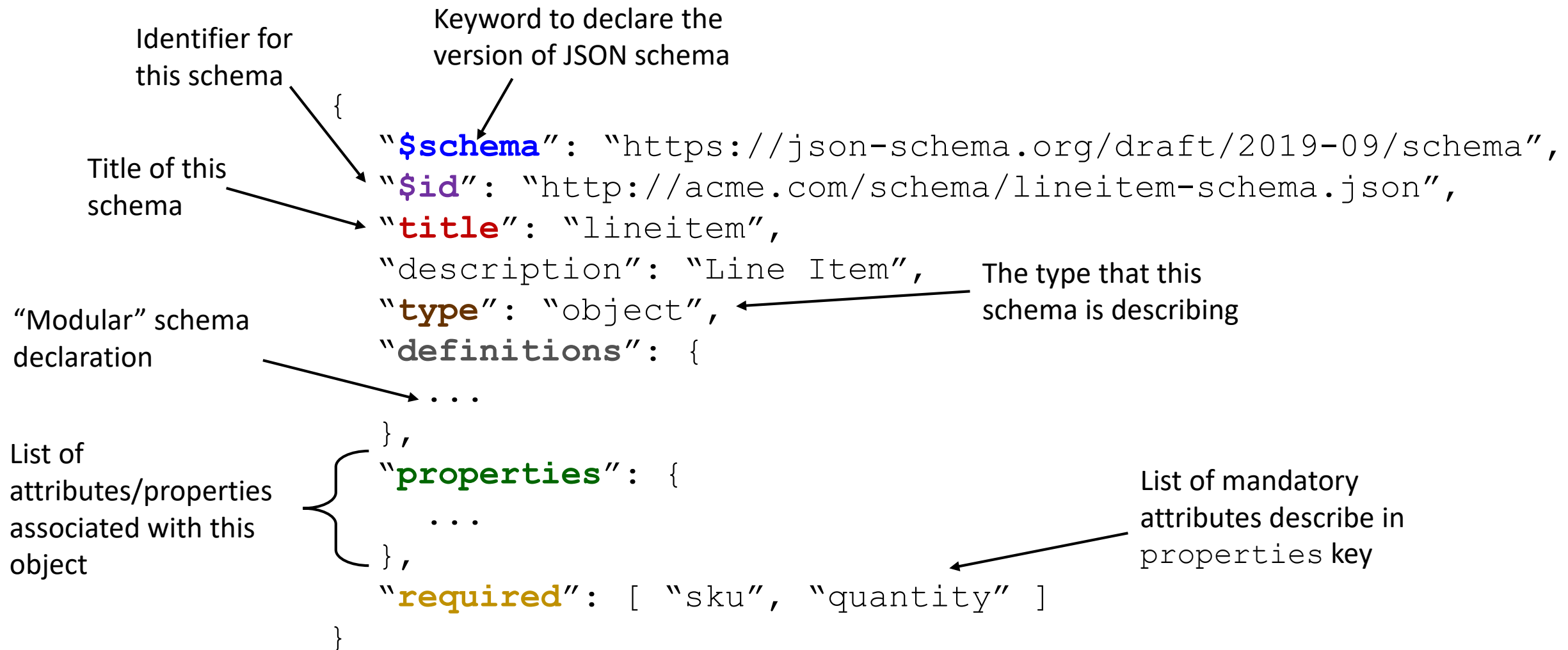 - Used by the receiver to validate the data before consuming it

Why Use a Schema?

JSON document

JSON document is produced according to a specific schema

Schema

Schema

3rd Party Applications

Validate the received data is syntactically and structurally correct before consuming it

JSON Schema

- For annotating and validating JSON documents
  - Standard https://json-schema.org
- Externalize the structure of a JSON documents
- Use tools to validate JSON against its schema
  - Multiple tools for different environment (Java, Go, PHP, etc)
  - Allow JSON document to created and received operated correctly
- MongoDB supports a subset of JSON Schema
  - Can be used to enforce the structure of documents when they're inserted or updated
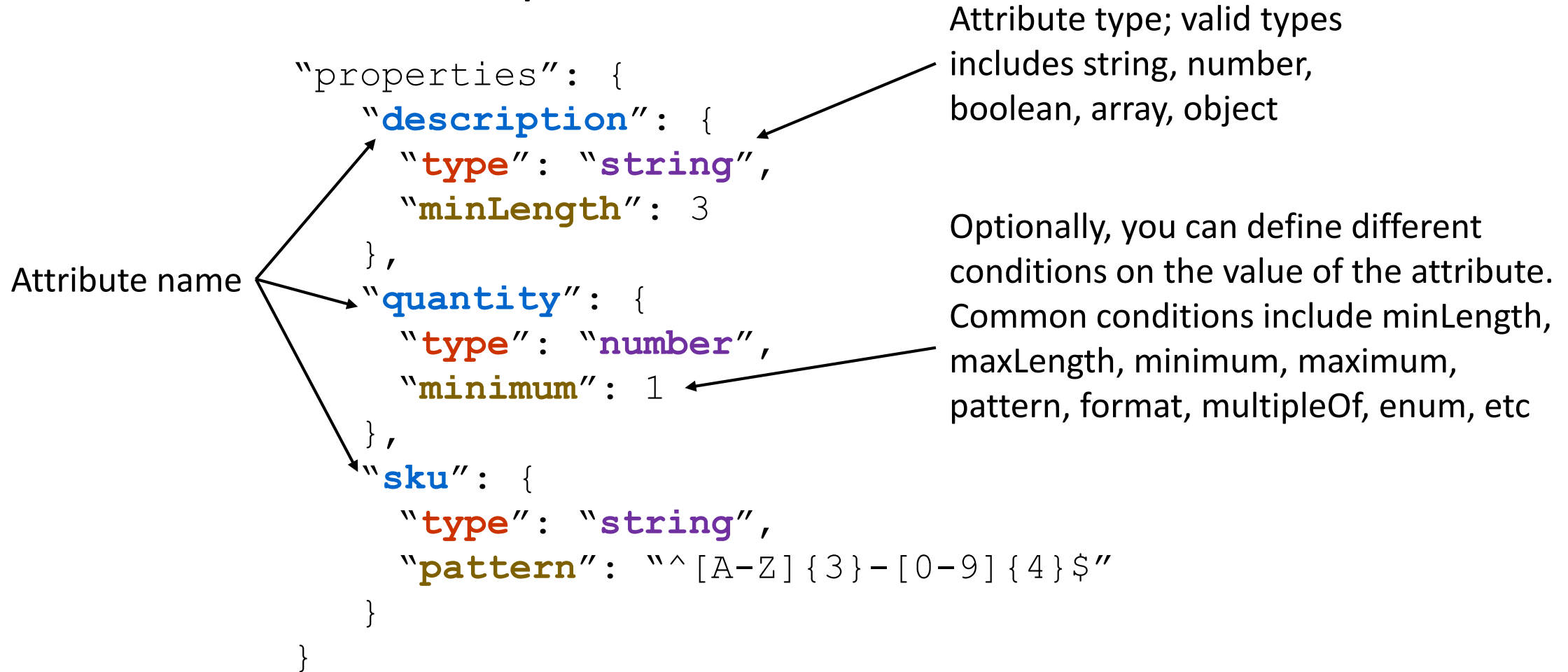- Human and machine readable

Basic Structure of a Schema

Identifier for this schema

Keyword to declare the version of JSON schema

Title of this schema

"Modular" schema declaration

List of attributes/properties associated with this object

The type that this schema is describing

List of mandatory attributes describe in `properties` key

```json
{
    "$schema": "https://json-schema.org/draft/2019-09/schema",
    "$id": "http://acme.com/schema/lineitem-schema.json",
    "title": "lineitem",
    "description": "Line Item",
    "type": "object",
    "definitions": {
        ...
    },
    "properties": {
        ...
    },
    "required": [ "sku", "quantity" ]
}
```

Schema Example

```
"properties": {
    "description": {
        "type": "string",
        "minLength": 3
    },
    "quantity": {
        "type": "number",
        "minimum": 1
    },
    "sku": {
        "type": "string",
        "pattern": "^[A-Z]{3}-[0-9]{4}$"
    }
}
```

Attribute name

Attribute type; valid types includes string, number, boolean, array, object

Optionally, you can define different conditions on the value of the attribute. Common conditions include minLength, maxLength, minimum, maximum, pattern, format, multipleOf, enum, etc

See https://json-schema.org/understanding-json-schema/reference/index.html for more details

Schema Example

```json
"definitions": {
    "lineitem": { ... }
},
"type": "object",
"properties": {
    "orderDate": {
        "type": "string",
        "format": "date"
    },
    "rush": {
        "type": "boolean",
        "default": "false"
    },
    "details": {
        "type": "array",
        "items": {
            "$ref": "#/definitions/lineitem"
        }
    }
},
```

Schema of `lineitem`

Build in format: date, date-time, email, ipv4, url, etc

Specify default values

Set the element type of an array

Special keyword to denote a reference

Schema of `lineitem` in definitions

Example Schema of Response Envelope

```
{
    "$schema": " https://json-schema.org/draft/2019-09/schema",
    "$id": "http://acme.com/response-envelope.json",
    "title": "Response Envelope",
    "type": "object",
    "properties": {
        "status": {
            "type": "integer",
            "minimum": 100,
            "maximum": 599
        },
        "message": {
            "type": "string"
        },
        "timestamp": {
            "type": "string",
            "format": "date-time"
        },

                                            "version": {
                                              "type": "string",
                                              "enum": [ "v1", "v2" ]
                                            },
                                            "data": {
                                              "type": "string"
                                            },
                                            "action": {
                                              "$ref": "#/definitions/actions"
                                            }
                                        },
                                        "required" [ "status", "timestamp", "version" ]
                                    }
    }
}
```

Example Validation with Schema – 1

```
const { Validator, ValidationError } = require('express-json-validator-middleware');
const validator = new Validator({ allErrors: true });
const bodyParser = require('body-parser');

const respSchema = require('./schema/response.json');
const customerSchema = require('./schema/customer.json');

app.post('/customer', bodyParser.json(),
    validator.validate({ body: customerSchema }),
    (req, resp) => {
        ...
    }
);

app.use((err, req, resp) => {
    if (err instanceof ValidationError) {
        //Handle error
        resp.status(400);
        ...
    }
});
```

Create a new instance of Validator

Validating a JSON payload. Don't forget to parse the payload to JSON first .
If validation fails, the request will be passed to the next handler

Use Express error handler to handle validation error.

Check if the error is a schema validation error. If it is handle it. Specific errors are listed in `err.validations`

Example Validation with Schema – 2

```
app.post('/customer', bodyParser.json(),
   expValidate({ body: customerSchema }),
   (req, resp) => {
      const customer = req.body;
      //Do something with customer
      ...
      const resp = {
         status: 201,
         data: 'Data has been processed',
         timestamp: (new Date()).toISOString(),
         version: 'v1'
      }
      const valid = validator.ajv.validate(respSchema, resp);
      if (!valid)
         console.error(validator.ajv.errors);
      resp.status(201)
      resp.json(resp);
   }
);
```

Validate the response before sending out.
Note: this should be done in unit testing.
A better use would be to validate responses
from REST APIs

Log the validation error

JSON Schema to Other Languages

- Lots of tools to generate JSON schema to data structure in specific languages
  - Java/Scala - http://www.jsonschema2pojo.org
  - Python - https://github.com/cwacek/python-jsonschema-objects
  - TypeScript - https://github.com/bcherny/json-schema-to-typescript

What is OpenAPI?
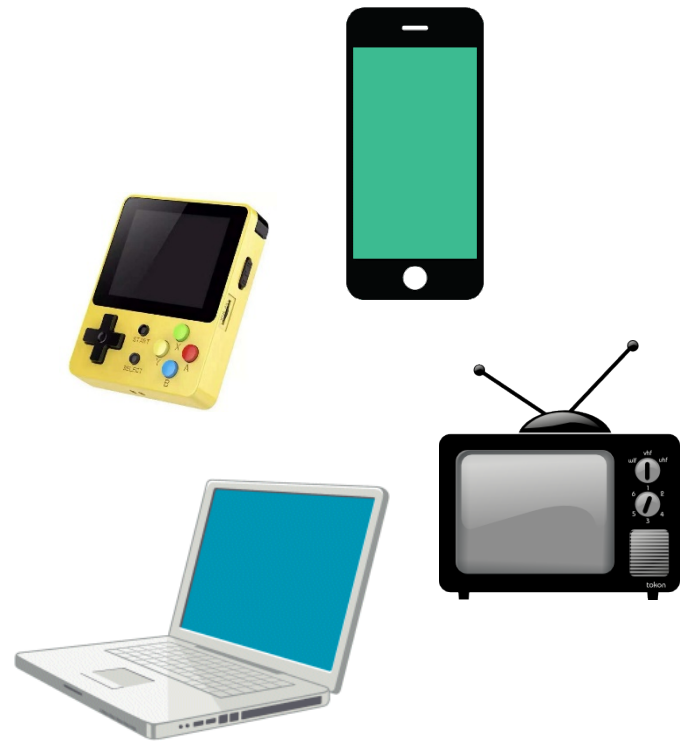
OpenAPI defines a contract between the API consumers and API producers

`POST /products`

`201 Created`

API Consumers

API Implementation

What is OpenAPI?

- A public contract of your RESTful API
- Machine readable
  - Written in either YAML or JSON
- Start with the specification of the API
  - Then implement it in a specific language
- Purpose
  - Documentation of the API - in a language agnostic way
  - Enforce validity of invoking a request and the returned response
  - Used as a tool to generate servers and clients
- Integrates with JSON Schema
  - Ensures that the data exchanged between the client and server conforms to an agreed upon standard

OpenAPI

- Contains
  - Info - Basic information about the the API
  - Servers - Describe the base URL of the API
  - Path - list the valid HTTP methods and resources.

    Also includes request parameter and responses

    Security - supported security mechanism
  - Components - common objects that can be reused
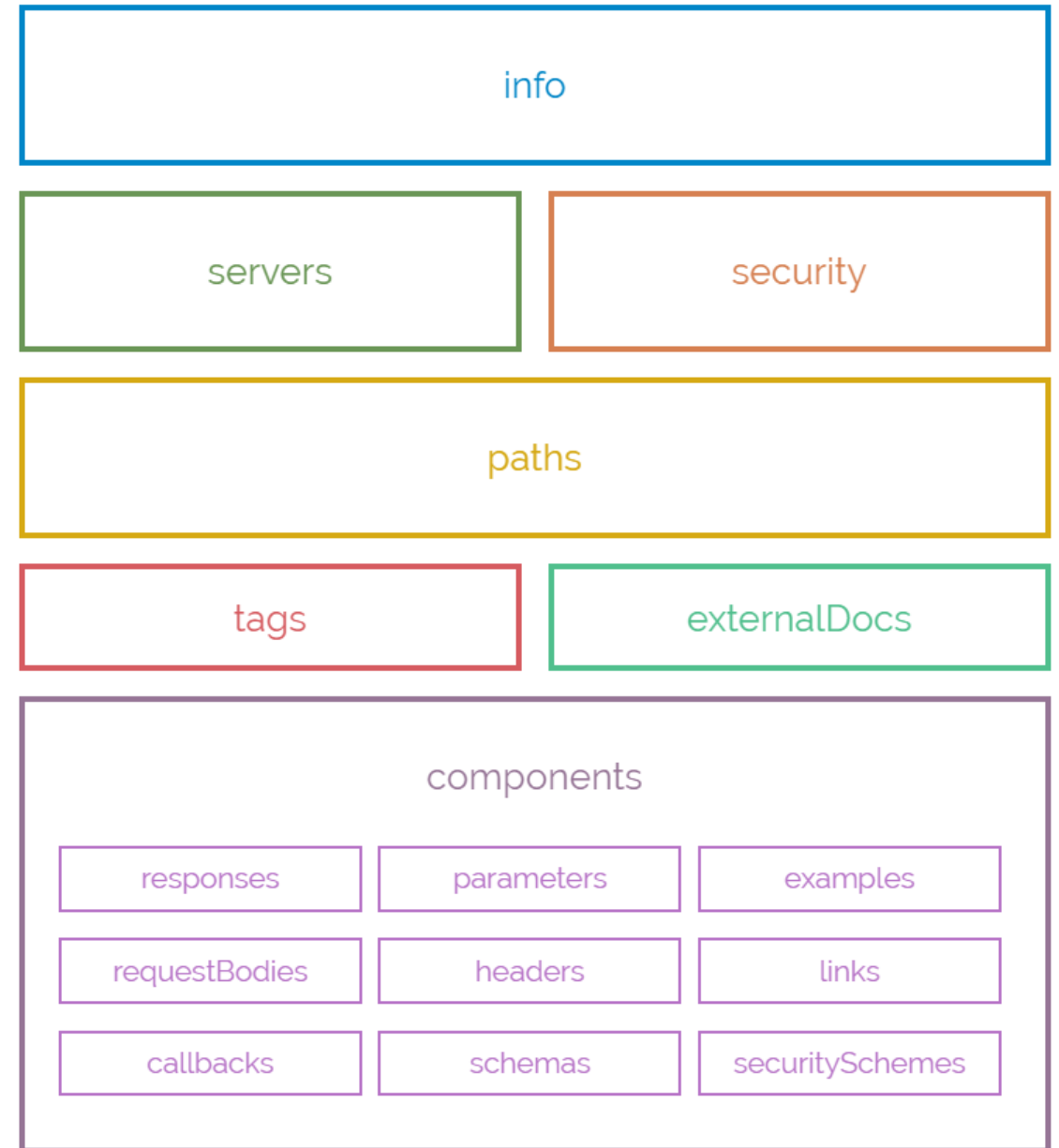
See https://swagger.io/specification/ for more details

Example OpenAPI

- Endpoints/resources

    `/api/customers`

- Permissible methods for each end point

    `GET`

- Parameters

    `/api/customers?limit=20&offset=10`

- Responses

    `200 OK`

- Content type
    - As defined in a JSON schema

OpenAPI Example

```javascript
const qs = new HttpParams()
    .set('limit', '20')
    .set('offset', '10');

this.http.get('http://acme.com/api/customers', { params: qs })
  .toPromise()
  .then(result => { /* do something with result */ })
```

```html
<form method="GET" action="/api/customers">

  <input type="number" name="limit">
  <input type="number" name="offset">

  <button type="submit">Go</button>
</form>
```

Example OpenAPI

OpenAPI version

```yaml
openapi: "3.0.3"
info:
  title: "Customer"
  description: "Manage customers"
  version: "v1.0"
servers:
- url: https://dev.acme.com
  description: "Development server"
- url: https://api.acme.com
  description: "Production server"
```
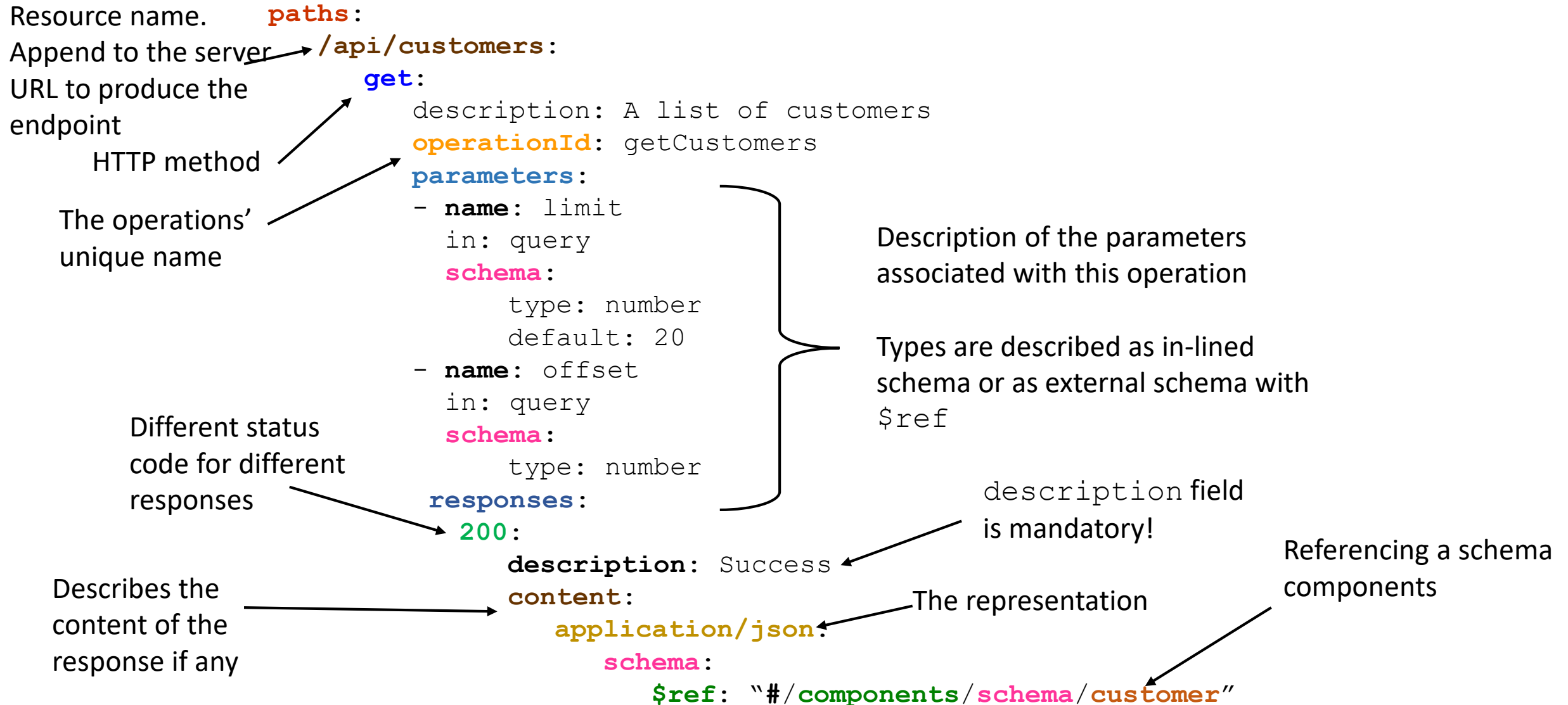
Information about the API

Provides connectivity information about the API server

Example OpenAPI

Resource name. Append to the server URL to produce the endpoint

HTTP method

The operations' unique name

The operations' unique name

Different status code for different responses

Describes the content of the response if any

Description of the parameters associated with this operation

Types are described as in-lined schema or as external schema with `$ref`

`description` field is mandatory!

The representation

Referencing a schema components

```yaml
paths:
  /api/customers:
    get:
      description: A list of customers
      operationId: getCustomers
      parameters:
      - name: limit
        in: query
        schema:
          type: number
          default: 20
      - name: offset
        in: query
        schema:
          type: number
      responses:
        200:
          description: Success
          content:
            application/json:
              schema:
                $ref: "#/components/schema/customer"
```

Example OpenAPI

```
paths:
    /api/customer:
        get:
            parameters:
```
Referencing parameter types defined in components
```
            - $ref: "#/components/parameters/limit"
            - $ref: "#/components/parameters/offset"
            ...
```

Reusable objects are placed under components. They are referenced with $ref
```
components:
    parameters:
        limit:
            name: limit
            in: query
            description: "Limit the number of return results"
            schema:
                type: number
                default: 20
    schema:
        customer:
            type: object
            ...
```

Parameter Types

Type	Example	Result
Query String	```/customers
get:
parameters:
- name: offset
in: query```	`GET /customers?offset=10`
Path	```/customer/{custId}
get:
parameters:
- name: custId
in: path```	`GET /customer/12`
Header	```/customers
get:
parameters:
- name: X-ApiKey
in: header```	```GET /customers
X-ApiKey: abc123```

OpenAPI Example

```
const qs = new HttpParams()
      .set('name', 'fred')
      .set('email', 'fred@bedrock.com');
const headers = new HttpHeaders()
      .set('Content-Type', 'application/x-www-form-urlencoded')

this.http.post('http://acme.com/api/customer', params.toString(),
    { params: qs })
    .toPromise()
    .then(result => { /* do something with result */ })
```

```
<form method="POST" action="/api/customer">

   <input type="text" name="name">
   <input type="email" name="email">
   ...
   <button type="submit">Go</button>
</form>
```
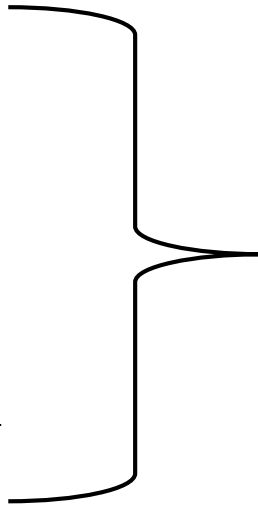
Example OpenAPI

```
paths:
   /api/customer:
      post:
         description: Add a new customer
         operationId: addCustomer
         requestBody:
            content:
               'application/x-www-form-urlencoded':
                  schema:
                     type: object
                     properties:
                        name:
                           type: string
                        email:
                           type: string
                           format: email
                           ...
```

Describe the content of the request' body

Schema for the request payload. Can be defined externally with a `$ref`

Ensuring Implementation Compliance

```
const OpenAPIValidator = require('express-openapi-validator).OpenApiValidator;
const bodyParser = require('body-parser')
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true });

async new OpenApiValidator(
    { apiSpec: __dirname + '/assets/customers-api.yaml'})
    .install(app);

app.get('/customer', (req, resp) => {
    ...
})


app.use((err, req, resp, next) => {
    if (err.status) {
        //handle error
})
```

Validated by OpenAPI

Load the OpenAPI specification. Any route after this will be validated by OpenAPI

Install the OpenAPI validator on the Express application

Only request that conforms to the specification are processed. Errors will be reported

Ensuring Implementation Compliance

```
const OpenAPIValidator = require('express-openapi-validator).OpenApiValidator;
const bodyParser = require('body-parser')
const app = express();


app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

async new OpenApiValidator(
    { apiSpec: __dirname + '/assets/customers-api.yaml'})
    .install(app);

app.get('/customer', (req, resp) => {
    ...
})


app.use((err, req, resp, next) => {
    if (err.status) {
        //handle error
    }
})
```

Load the OpenAPI specification

Install the OpenAPI validator on the Express application

Only request that conforms to the specification are processed. Errors will be reported

OpenAPI First

Generate server skeleton

OpenAPI specification of RESTful API

Generate client stubs

Communicate
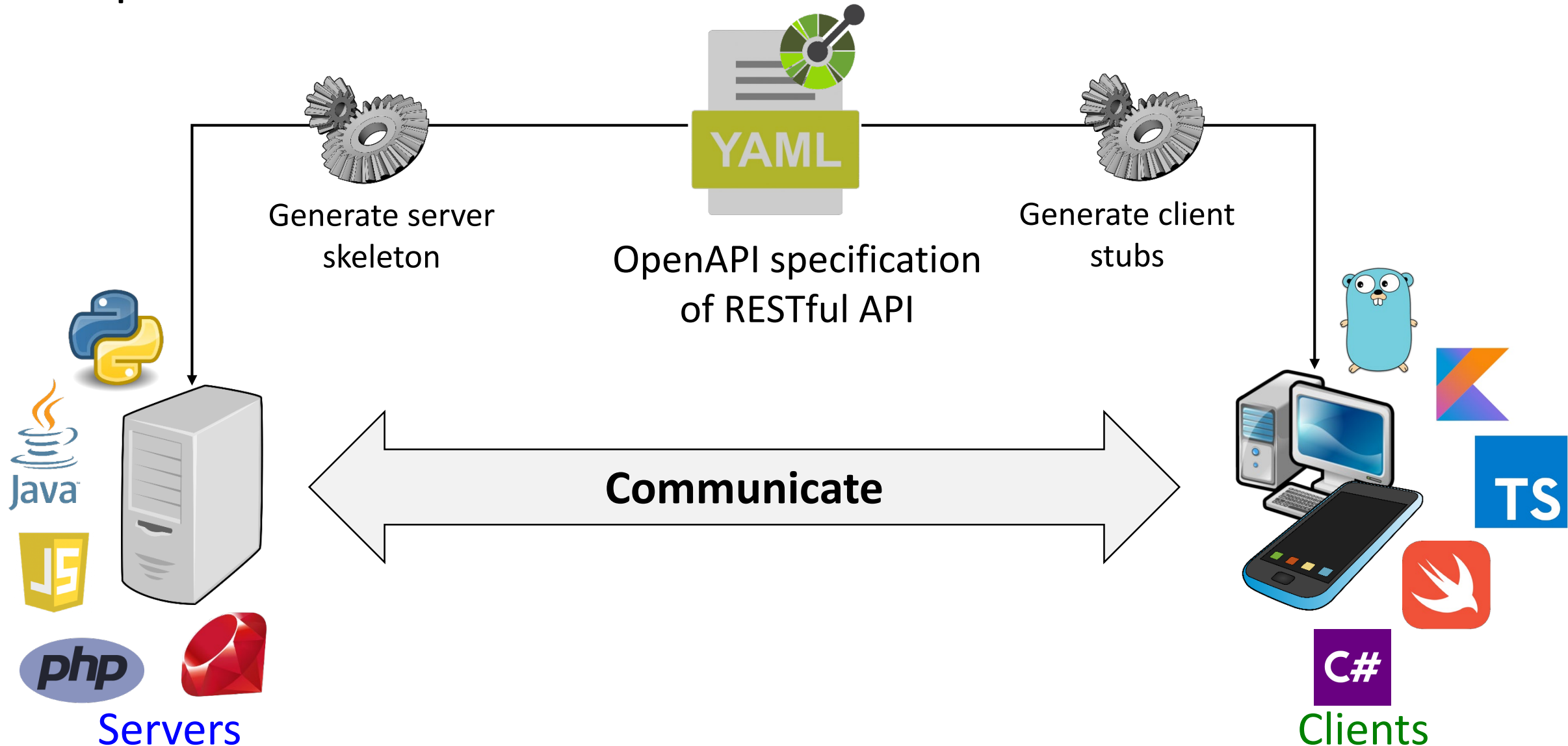
Servers

Clients

OpenAPI Generator

- Use OpenAPI specification as a tool to generate
  - Server skeletons - API producers
  - Client stubs - API consumers
- Ensures that clients and servers can communicate
  - Takes care of the plumbing
- OpenAPI Tools supports range of programming languages and frameworks
  - Eg. typescript-angular, jaxrs-spec, nodejs-server, php,

OpenAPI Generator

- Installing

```
npm install -g @openapitools/openapi-generator-cli
```

- Generating Node server in TypeScript

```
openapi-generator generate -i myapi.yaml \
         -g typescript-node -o src/server
```

- Generating Angular client

```
openapi-generator generate -i myapi.yaml \
         -g typescript-angular -o src/client
```

List of Modules

- express-json-validator-middleware - https://www.npmjs.com/package/express-json-validator-middleware

- express-openapi-validator - https://www.npmjs.com/package/express-openapi-validator

Criticism with HTTP API Style

- Multiple endpoints for data
  - There is a single entry point
  - But need to navigate to the required data with multiple invocation
- Over or under fetch data
  - Data returned is predetermined by the developers
- Data aggregation
  - Data aggregation involves multiple invocation to the service
  - Client performs the integration
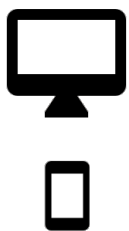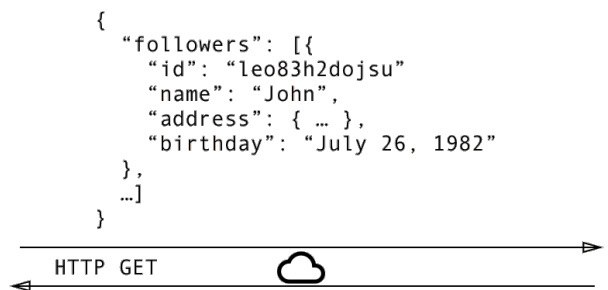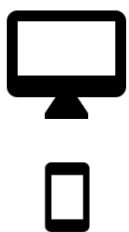- Schema and OpenAPI are add-ons
  - Not integral part of HTTP APIs
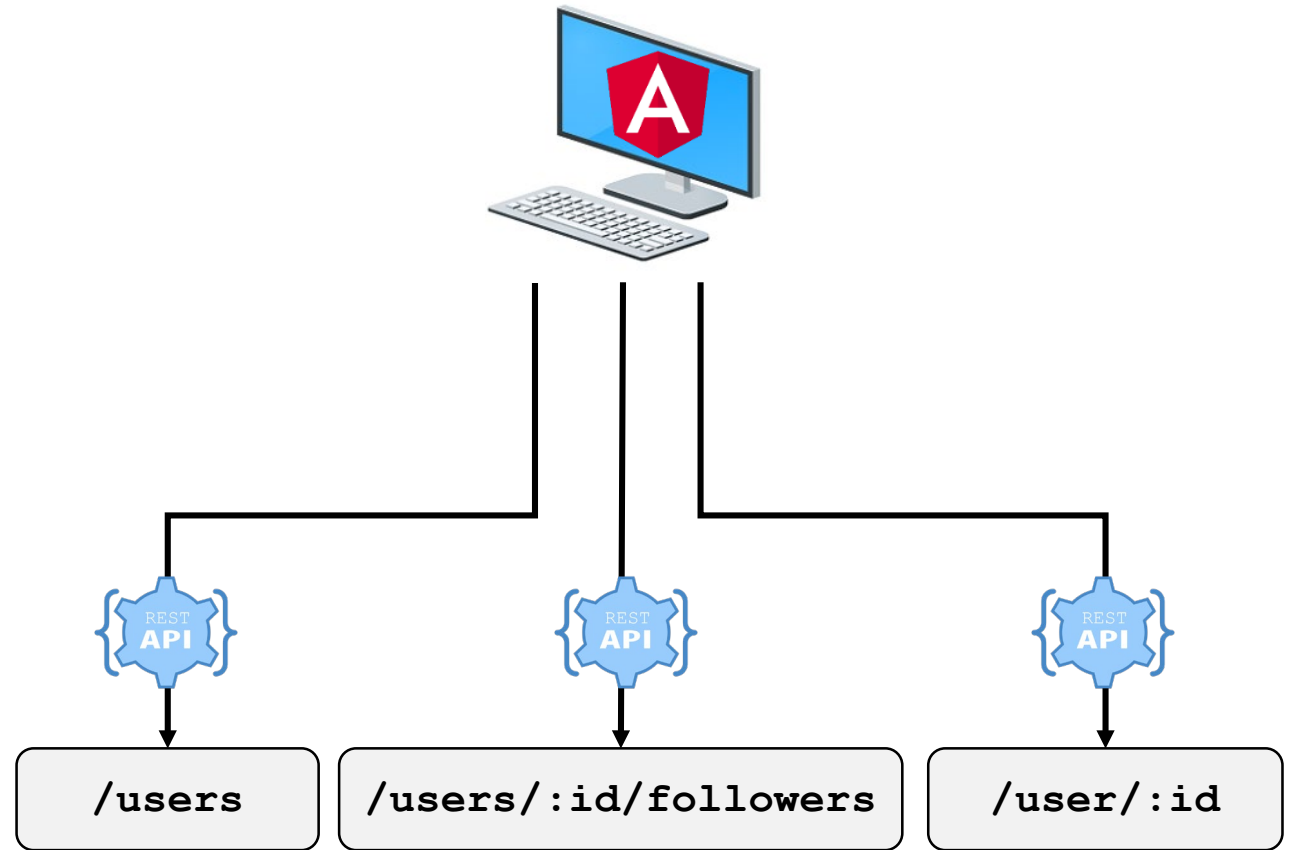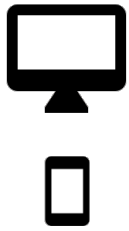
`/users`

`/users/:id/followers`

`/user/:id`

REST API

1

```
HTTP GET

{
  "user": {
    "id": "er3tg439frjw"
    "name": "Mary",
    "address": { … },
    "birthday": "July 26, 1982"
  }
}
```

/users/<id>

/users/<id>/posts

/users/<id>/followers

2

```
HTTP GET

{
  "posts": [{
    "id": "ncwon3ce89hs"
    "title": "Learn GraphQL today",
    "content": "Lorem ipsum … ",
    "comments": [ … ],
  }]
}
```

/users/<id>

/users/<id>/posts

/users/<id>/followers

3

```
{
  "followers": [{
    "id": "leo83h2dojsu"
    "name": "John",
    "address": { … },
    "birthday": "July 26, 1982"
  },
  …]
}

HTTP GET
```

/users/<id>

/users/<id>/posts

/users/<id>/followers

Image from https://i.imgur.com/VRyV7Jh.png

What is GraphQL?

- Query language and a runtime engine to process the query and return the appropriate data

- Query against a predefined schema
  - Like relational databases

- Client driven data access
  - Places the responsibility of the required data in hands of the consumer
  - Flexibility to query any data in a give schema given the right credentials

- No fixed 'API', so easier to evolve the underlying data

GraphQL