



UNIVERSITÀ
DI TORINO

Documenting a RESTFUL Server

Prof. Fabio Ciravegna
Dipartimento di Informatica
Università di Torino
fabio.ciravegna@unito.it



RESTFUL Service

- A Restful service is defined by the following characteristics
 - Client Server communication
 - Stateless
 - as in HTTP-based communication
 - Cacheable
 - responses can be cached
 - Layered
 - it can communicate via a proxy

RESTFUL Web Services

- Defined by:
 - A request:
 - An Endpoint URL
 - it will define one or more endpoints (URLS) with a **domain**, **port**, **path**,
 - and/or **querystring** (e.g. `http://mydomain.org:3000/map/183?format=json`)
 - **or** A body containing the parameters (typically if POST)
 - An HTTP method (CRUD)
 - CRUD: Create, read, update, delete
 - GET, POST, PUT, DELETE
 - HTTP headers
 - we have covered those in the HTTP lecture
 - A response: a payload containing
 - a status code and a set of data

Documenting the API

- The API is your connection with the rest of the world
 - it is the way the others can connect to your servers
- A precise and formal documentation is necessary in order to
 - provide information about how to connect to your servers
 - provide confirmation of service and meeting of requirements for a customer
- The documentation must be designed for whomever **has no access to the code**
 - i.e. it must be self standing, documenting the connections with the real world (parameters in and out) as well as the expected behaviour
 - it is **not** a way to document the internal code!!!



UNIVERSITÀ
DI TORINO

Documenting an API using OpenAPI3.0

Prof. Fabio Ciravegna
Department of Computer Science
The University of Sheffield
f.ciravegna@shef.ac.uk



© Prof. Fabio Ciravegna, Università di Torino

Open API

- The OpenAPI Specification is a community-driven open specification
 - within the OpenAPI Initiative, a Linux Foundation Collaborative Project.
- Its Specification (OAS) defines a standard, programming language-agnostic interface description for HTTP APIs
 - it allows **both** humans and computers to discover and understand the capabilities of a service
 - without requiring access to source code, additional documentation, or inspection of network traffic
 - When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

ctd

- The OpenAPI Specification does not require rewriting existing APIs
- It does not require binding any software to a service
 - the service being described may not even be owned by the creator of its description
- It does, however, require the capabilities of the service to be described in the structure of the OpenAPI Specification

Definitions

- OpenAPI Document
 - A document (or set of documents) that defines or describes an API. An OpenAPI definition uses and conforms to the OpenAPI Specification
- Path Templating (i.e. Route Templating)
 - Path templating refers to the usage of template expressions, delimited by curly braces (`{}`), to mark a section of a URL path as replaceable using path parameters
 - Each template expression in the path **MUST** correspond to a path parameter that is included in the Path Item itself and/or in each of the Path Item's Operations

- Media Types

- Media type definitions are spread across several resources.
Examples of possible media type definitions:

- `text/plain; charset=utf-8`
- `application/json`

- HTTP Status Codes

- The HTTP Status Codes are used to indicate the status of the executed operation

•

Basic Information

<https://medium.com/wolox/documenting-a-nodejs-rest-api>

- the basic information for our project, such as title, version, description, and other things.

```
{
  openapi: '3.0.1',
  info: {
    version: '1.3.0',
    title: 'Users',
    description: 'User management API',
    termsOfService: 'http://api_url/terms/',
    contact: {
      name: 'Wolox Team',
      email: 'hello@wolox.co',
      url: 'https://www.wolox.com.ar/'
    },
    license: {
      name: 'Apache 2.0',
      url: 'https://www.apache.org/licenses/LICENSE-2.0.html'
    }
  },
};
```

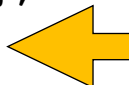
Version of OpenApi

Version of your software/documentation

Service title/description/terms of service

Contacts

Software licence



Insert here the code in the next slides

Servers

- You can have different entry points
 - e.g. depending on being developing or in production

```
{  
  /* ... */  
  servers: [  
    {  
      url: 'http://localhost:3000/',  
      description: 'Local server'  
    },  
    {  
      url: 'https://api_url_testing',  
      description: 'Testing server'  
    },  
    {  
      url: 'https://api_url_production',  
      description: 'Production server'  
    }  
  ],  
  /* ... */  
};
```

if we declare different servers, we are declaring that the documentation below is relevant to all of them

← Insert here the code in the next slides

Endpoint documentation

- This is key to the documentation
- It describes for each endpoint:
 - the operation (Get, Post...)
 - the parameters
 - where to find it (header, query, body, ...)
 - the type (e.g. integer or an object with specific fields)
 - any default value (e.g. 1)
 - required (yes/no)
- The responses
 - https value returned (e.g. 200, 404,...) (so it declares success or failure)
 - the type of data returned (e.g. application/json)
 - the data structure describing the data (e.g. an array of objects with specific fields)



Example

The tags the route is tagged with

The list of parameters

type is json

Responses

Error

Returned object

Example

The route

The type (post)

These parameters are to be found in the body (e.g. via bodyParser in node)

type definition (see next slide, grey arrow)

Success (200)

Type definition (next slide, orange arrow)

```
{
  /* ... */
  paths: {
    '/users': {
      /* ... */
      post: {
        tags: ['CRUD operations'],
        description: 'Create users',
        operationId: 'createUsers',
        parameters: [],
        requestBody: {
          content: {
            'application/json': {
              schema: {
                $ref: '#/components/schemas/Users'
              },
              required: true
            },
          },
          responses: {
            '200': {
              description: 'New users were created'
            },
            '400': {
              description: 'Invalid parameters',
              content: {
                'application/json': {
                  schema: {
                    $ref: '#/components/schemas/Error'
                  },
                  example: {
                    message: 'User identificationNumbers 10, 20 already exist',
                    internal_code: 'invalid_parameters'
                  }
                }
              }
            }
          }
        }
      }
    }
  }
};
/* ... */
};
```

```
{
  /* ... */
  paths: {
    '/users': {
      /* ... */
      post: {
        tags: ['CRUD operations'],
        description: 'Create users',
        operationId: 'createUsers',
        parameters: [],
        requestBody: {
          content: {
            'application/json': {
              schema: {
                $ref: '#/components/schemas/Users'
              }
            },
            required: true
          },
          responses: {
            '200': {
              description: 'New users were created'
            },
            '400': {
              description: 'Invalid parameters',
              content: {
                'application/json': {
                  schema: {
                    $ref: '#/components/schemas/Error'
                  },
                  example: {
                    message: 'User identificationNumbers 10, 20 already exist',
                    internal_code: 'invalid_parameters'
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  /* ... */
};
```

Grouping the end points

- When you have a large number of end points it may be confusing to remember the role of each one in the grand schema of things
- The tag TAGS allows to declare the groups they are divided into
 - in the code you will use different routes

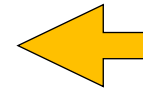
```
.    {  
      /* ... */  
      tags: [  
        {  
          name: 'CRUD operations'  
        }  
      ],  
      /* ... */  
    };
```

It is an array - list all the allowed tags here

\$ref

- Often you will have to define a schema that is used/received and/or returned by several routes
- You can write it just once and then refer it via \$ref

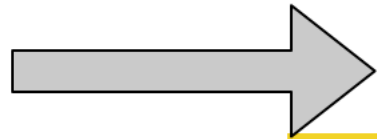
```
requestBody: {  
  content: {  
    'application/json': {  
      schema: {  
        $ref: '#/components/schemas/Users'  
      }  
    }  
  },  
  required: true  
},
```



- The definition is found in the file Users (next slide)

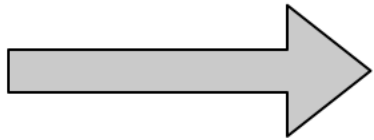


Users



Type User

Properties of types
defined above



Type Users

Users is an Array of Type User



Error type

```

{
  /* ... */
  components: {
    schemas: {
      identificationNumber: {
        type: 'integer',
        description: 'User identification number',
        example: 1234
      },
      username: {type: 'string', example: 'raparicio'},
      userType: {type: 'string', enum: USER_TYPES, default: REGULAR},
      companyId: {type: 'integer', description: 'Company id where the user e
    User: {
      type: 'object',
      properties: {identificationNumber: {
        $ref: '#/components/schemas/identificationNumber'},
        username: {$ref: '#/components/schemas/username'},
        userType: {$ref: '#/components/schemas/userType'},
        companyId: {$ref: '#/components/schemas/companyId'}
      }
    },
    Users: {
      type: 'object',
      properties:
        {users: {type: 'array', items: {$ref: '#/components/schemas/User'}}
      },
    Error: {
      type: 'object',
      properties: {
        message: {
          type: 'string'
        },
        internal_code: {
          type: 'string'
        }
      }
    }
  }
};

```

You refer to the same file
giving the complete path

Users

```
{
  /* ... */
  components: {
    schemas: {
      identificationNumber: {
        type: 'integer',
        description: 'User identification number',
        example: 1234
      },
      username: {type: 'string', example: 'raparicio'},
      userType: {type: 'string', enum: USER_TYPES, default: REGULAR},
      companyId: {type: 'integer', description: 'Company id where the user example: 15},
      User: {
        type: 'object',
        properties: {identificationNumber: {
          $ref: '#/components/schemas/identificationNumber'},
          username: {$ref: '#/components/schemas/username' },
          userType: {$ref: '#/components/schemas/userType' },
          companyId: {$ref: '#/components/schemas/companyId'}
        }
      },
      Users: {
        type: 'object',
        properties: {
          users: {type: 'array', items: {$ref: '#/components/schemas/User'}}
        }
      },
      Error: {
        type: 'object',
        properties: {
          message: {
            type: 'string'
          },
          internal_code: {
            type: 'string'
          }
        }
      }
    }
  }
};
```

file: '#/components/schemas/Users'

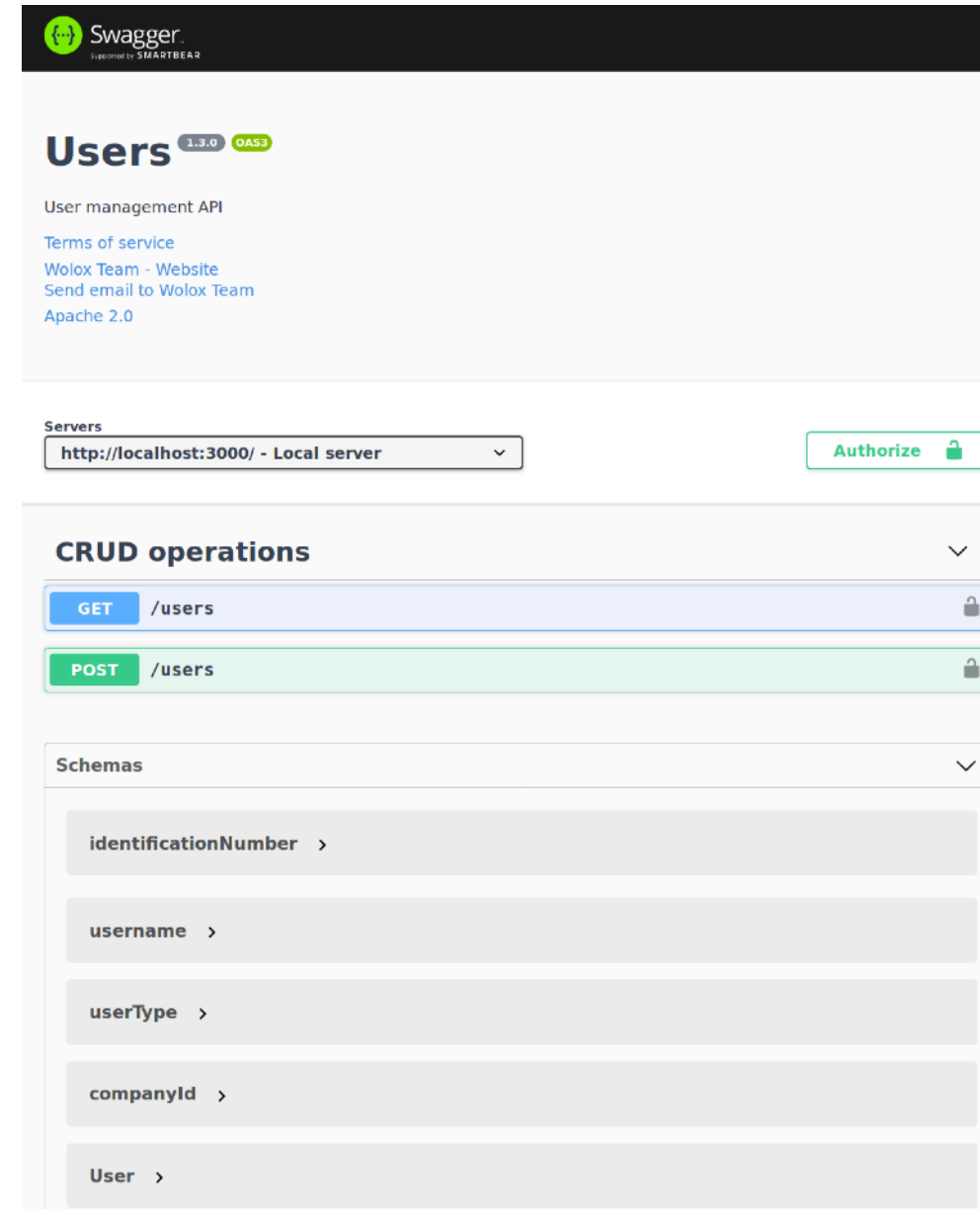
repeated from previous slide (not an image)

Integration with NodeJS

- install the package swagger-ui-express
 - `npm i swagger-ui-express --save`
- In the app.js file
 - `const swaggerUi = require('swagger-ui-express');`
 - `const openApiDocumentation = require('./openApiDocumentation');`
- Declare the route where the documentation can be seen
 - e.g. /api-docs
 - `app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(openApiDocumentation));`
 - NOTE: this route is only described in **app.js**
 - it is NOT described in the Express route files

Accessing the documentation

- Start the server and go to the documentation route /api-docs
 - e.g. <http://localhost:3000/api-docs>

A screenshot of the Swagger UI for a 'Users' API. The header shows the Swagger logo and 'Supported by SMARTBEAR'. The main title is 'Users' with version '1.3.0' and 'OAS3' tags. Below the title, it says 'User management API' and provides links for 'Terms of service', 'Wolox Team - Website', 'Send email to Wolox Team', and 'Apache 2.0'. A 'Servers' dropdown menu is set to 'http://localhost:3000/ - Local server', with an 'Authorize' button. The 'CRUD operations' section shows a 'GET /users' operation in blue and a 'POST /users' operation in green. The 'Schemas' section lists 'identificationNumber', 'username', 'userType', 'companyId', and 'User', each with a right-pointing arrow.

Endpoint doc for GET /users

GET
/users

Get users

Parameters
Try it out

Name	Description
x-company-id * required integer (header)	Company id where the users work
page integer (query)	Default value : 1
orderBy string (query)	Available values : asc, desc Default value : asc

Responses

Code	Description	Links
200	<div>Users were obtained</div> <div> application/json </div> <div>Controls Accept header.</div> <div> Example Value Schema </div> <pre> { "users": [{ "identificationNumber": 1234, "username": "raparicio", "userType": "regular", "companyId": 15 }] } </pre>	No links
400	<div>Missing parameters</div> <div> application/json </div> <div>Example Value Schema</div>	No links

Doc for Post to /users

POST /users

Create users

Parameters [Try it out](#)

No parameters

Request body required application/json

Example Value | [Schema](#)

```
{
  "users": [
    {
      "identificationNumber": 1234,
      "username": "raparicio",
      "userType": "regular",
      "companyId": 15
    }
  ]
}
```

Responses

Code	Description	Links
200	<div>New users were created</div>	No links
400	<div>Invalid parameters</div>	No links

application/json

Example Value | [Schema](#)

```
{
  "message": "User identificationNumbers 10, 20 already exist",
  "internal_code": "invalid_parameters"
}
```




UNIVERSITÀ
DI TORINO

Documenting routes within the code

to generate swagger docs



From JSDoc to Swagger documents

- You can use a plugin called `swagger-jsdoc`
 - `npm install swagger-jsdoc`
- JSDoc entries will include Swagger-specific annotations
 - e.g., `@swagger`, `@path`, `@method`, etc.

```
/**
 * @swagger
 * /api/v1/users:
 *   get:
 *     summary: Retrieve a list of users
 *     responses:
 *       200:
 *         description: A list of users
 */
app.get('/api/v1/users', (req, res) => {
  res.json([...]);
});
```

- Configure swagger-jsdoc:
 - Add a configuration file to specify where to look for annotations, and run it to generate your documentation file in JSON format
 - that you will store in the project

```
* @swagger
* path:
*   /route/path:
*     http-method:           // e.g., get, post, put, delete
*     summary: Short description of the route
*     description: Detailed description of what this route does
*     tags:
*       - TagName           // Tags help group related endpoints in Sw
*     parameters:           // Parameters for the endpoint
*       - in: query / path / body / header
*         name: parameterName
*         required: true / false
*         schema:
*           type: string / integer / object / array
*           description: Explanation of the parameter
*     responses:
*       200:                // HTTP status code
*       description: Success description
*       content:
*         application/json:
*           schema:
*             type: array / object / string
*             items:         // Only for array response types
*               type: object
*             properties:
*               key:         // Property name
*               type: string / integer / boolean
```

```
* @swagger
* path:
*   /api/v1/users:
*     get:
*       summary: Retrieve a list of users
*       description: Returns a list of all users
*       tags:
*         - Users
*       responses:
*         200:
*           description: A JSON array of user objects
*           content:
*             application/json:
*               schema:
*                 type: array
*                 items:
*                   type: object
*                   properties:
*                     id:
*                       type: integer
*                       description: User ID
*                     name:
*                       type: string
*                       description: User name
*/
app.get('/api/v1/users', (req, res) => {
  res.json([{ id: 1, name: "John Doe" }]);
});
```

Post Route

```

swagger
* path:
*   /api/v1/users:
*     post:
*       summary: Create a new user
*       description: Adds a new user to the database
*       tags:
*         - Users
*       requestBody:
*         required: true
*         content:
*           application/json:
*             schema:
*               type: object
*               properties:
*                 name:
*                   type: string
*                   description: User's name
*                   example: "Jane Doe"
*                 age:
*                   type: integer
*                   description: User's age
*                   example: 28
*       responses:
*         201:
*           description: User created successfully
*           content:
*             application/json:
*               schema:
*                 type: object
*                 properties:
*                   id:
*                     type: integer
*                     description: User ID
*                   name:
*                     type: string
*                     description: User's name
* /
app.post('/api/v1/users', (req, res) => {
  const { name, age } = req.body;
  res.status(201).json({ id: 1, name, age });
});

```

Tags Overview

- **@swagger**

- Indicates the Swagger block in swagger-jsdoc.

- **path:**

- Specifies the endpoint path.

- **http-method:**

- Specifies the HTTP method (get, post, put, delete).

- **summary**

- Short summary of the endpoint.

- **description**

- Detailed description of the endpoint's purpose.

- **tags**

- Groups the endpoint in Swagger UI.

- **parameters**

- Defines query, path, or header parameters for the endpoint.

- **requestBody**

- Defines body parameters for methods like POST and PUT.

- **responses**

- Specifies possible response codes and their structure.

note the semicolons!!

To sum up

- You should concentrate your efforts to provide high quality documentation to your APIs
 - Always remember to change the documentation if you change the code
- This will save time to you and would provide clarity to your users and co-workers