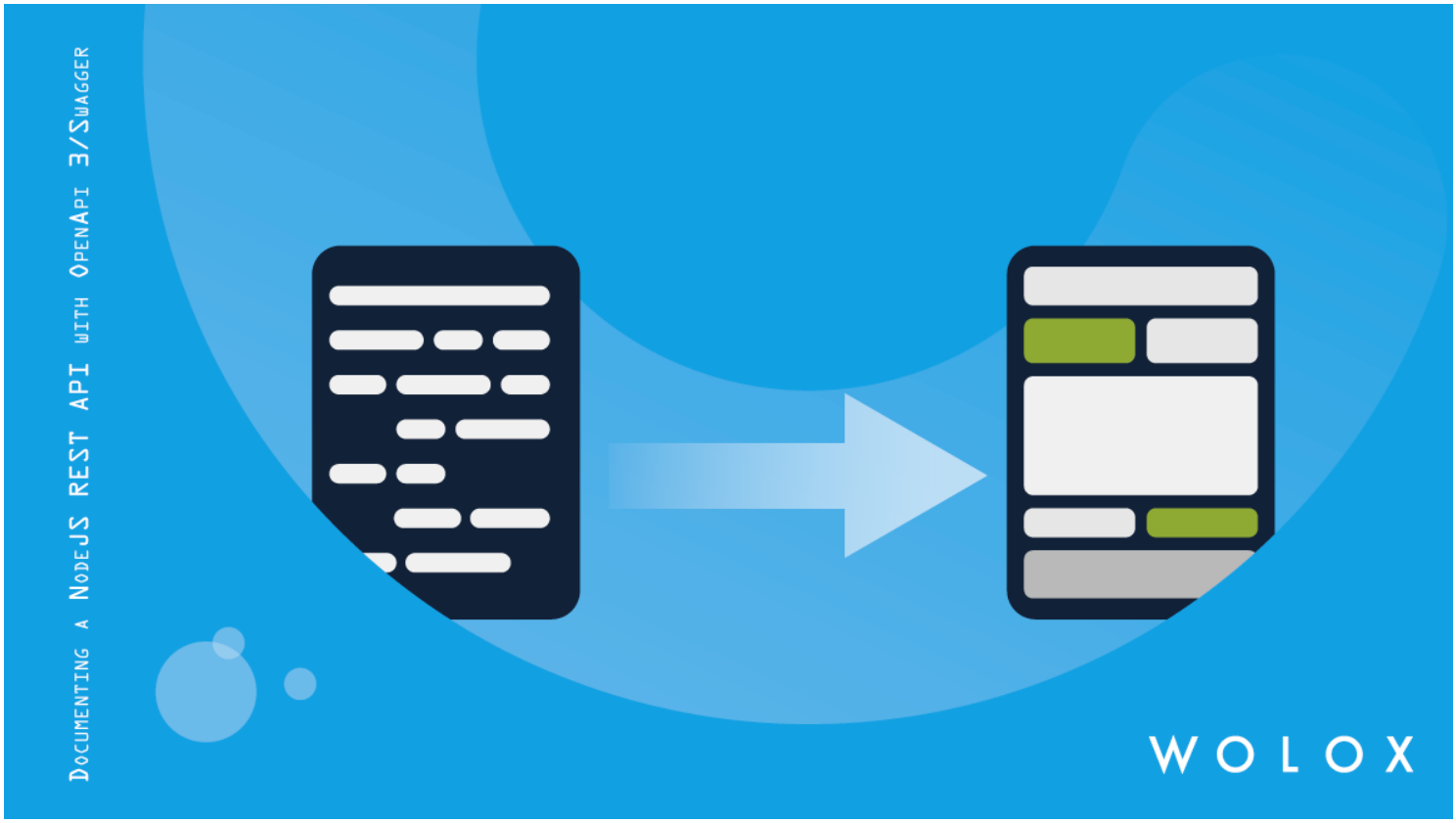


Documenting a NodeJS REST API with OpenApi 3/Swagger

How to take your project to the next level with OpenApi3/Swagger.

Rodrigo Aparicio Follow
Apr 15, 2019 · 7 min read

Introduction

At Wolox we are developing plenty of projects. I am currently working on one of the most complex: a marketplace app. It has users, products, dispatch flows, finances, purchase orders and logistics modules, all involving 6 API's that interact with each other in a microservice architecture. We combined different technologies a key one being NodeJS.

The time had come to document our API's endpoints. Based on our research, the best option was by far OpenApi 3 (A.K.A Swagger).

Sound reasons behind using OpenApi 3

We all agree that documentation is necessary but we never stop to get it done. We prioritize moving forward with the code without worrying about explaining what we are doing.

Luckily, we have this great tool that allows us to start improving our code documentation.

Some advantages of using OpenApi 3 are:

- **Quick** to use. The properties that you need to complete in every endpoint are always the same and there are only a few. You can also frequently reuse *components* and the generic information is simple to complete.
- Fun. Yes, you will enjoy doing documentation due to its user-friendly system. I promise you will be hooked on documentation!
- There are a lot of possible features. From describing parameters to making authentication and authorization schemes.
- **Interactive UI**. This is outstanding for many reasons I will talk about later.
- Maintainable and easily extendable. This can be achieved by leveraging the components I mentioned above and taking advantage of the constants file/s used in your API.

Let's start!

We need to create the file where we will write the documentation. The format can be either the JSON (Javascript file) or YAML. In this example I will use the JSON format.

First of all, we will configure the basic information for our project, such as title, version, description, and other things.

```
1  {
2    openapi: '3.0.1'
```

```
  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'
    }
  },
  /* ... */
};
```

basicInformation.js hosted with ❤ by **GitHub** **view raw**

Then, we can define the servers where our API is deployed.

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

servers.js hosted with ❤ by **GitHub** **view raw**

Now we are going to describe our API's endpoints, but before we do this, we need to create tags, which are like different topics, to group our endpoints.

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

tags.js hosted with ❤ by **GitHub** **view raw**

```
{
  /* ... */
   paths: {
    '/users': {
      get: {
        tags: ['CRUD operations'],
        description: 'Get users',
        operationId: 'getUsers',
        parameters: [
          {
            name: 'x-company-id',
            in: 'header',
            schema: {
              $ref: '#/components/schemas/companyId'
            },
            required: true,
            description: 'Company id where the users work'
          },
          {
            name: 'page',
            in: 'query',
            schema: {
              type: 'integer',
              default: 1
            },
            required: false
          },
          {
```

```
            name: 'orderBy',
            in: 'query',
            schema: {
              type: 'string',
              enum: ['asc', 'desc'],
              default: 'asc'
            },
            required: false
          }
        ],
        responses: {
          '200': {
            description: 'Users were obtained',
            content: {
              'application/json': {
                schema: {
                  $ref: '#/components/schemas/Users'
                }
              }
            }
          },
          '400': {
            description: 'Missing parameters',
            content: {
              'application/json': {
                schema: {
                  $ref: '#/components/schemas/Error'
                },
                example: {
                  message: 'companyId is missing',
                  internal_code: 'missing_parameters'
                }
              }
            }
          }
        }
      }
    },
    /* ... */
};
```

getUsers.js hosted with ❤ by **GitHub** **view raw**

Let’s make a pause and look over what we have done.

First, we have the *parameters* array where we can point out all the values that our endpoint can receive. Using the property **in** you will specify the appropriate place of each parameter. There are four possible values: *path*, *query*, *header* and *cookie*. For the body we need to use something else that I will explain in a minute.

Then, we have the **schema**: the parameter template. You can define it in place, as the *page* parameter in the example, or you can create a **component**. This option is awesome since you can use the components any time you want and avoid repeating code.

Here is an example:

```
{
  /* ... */
  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 works',
        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'
            }
          }
        }
      }
    }
};
```

components.js hosted with ❤ by **GitHub** **view raw**

Note that you can even call a component within another component! By making the documentation shorter and more maintainable, components will definitely make your life easier.

Also, please pay attention to *userType* object. With the **enum** property I can specify the values that userType can accept. In order to do that, I use an array *USER_TYPES* which is

declared in the constants file of my API. In the file where I write the documentation, I just required the array from the constants file. This is awesome! I do not have to repeat code and if I change the content of the array in the future I do not have to worry about updating the documentation. The same happens with the **default** value of userType, which is another constant of my API.

Let's see one more example, now a POST request.

```
{
  /* ... */
  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'
                }
```

```
                }
              }
            }
          }
        }
      }
    },
    /* ... */
};
```

createUsers.js hosted with ❤ by **GitHub** **view raw**

Note that now we have the *requestBody* object. There we specify the body that we receive in the request.

Documenting authentication

To finish with the examples, I will show you how to add the authentication you use in your API.

First, we need to specify the API security type. There are many available choices: *http*, *apiKey*, *oauth2* or *openIdConnect*. In this API we use apiKey:

```
{
  /* ... */
  security: [
    {
      ApiKeyAuth: []
    }
  ],
  /* ... */
};
```

authentication.js hosted with ❤ by **GitHub** **view raw**

And in the components I define:

```
{
  /* ... */
  components: {
    /* ... */
    securitySchemes: {
```

```
      ApiKeyAuth: {
        type: 'apiKey',
        in: 'header',
        name: 'x-api-key'
      }
    }
  }
};
```

authenticationComponent.js hosted with ❤ by **GitHub** **view raw**

In this way you will let OpenApi know that every endpoint in the API requires a header with *x-api-key* as key.

Here you can see examples of the other types of security in YAML.

Integration with NodeJS

This is the last and easiest part. In this API we are using **Express.js** so I will explain how to visualize the documentation you have done using this framework.

I wrote the documentation in a file called *openApiDocumentation.js* which is in the project's root. I strongly recommend that you take a look at the architecture of our NodeJS applications in this article.

Then, we should install the package swagger-ui-express. Run *npm i swagger-ui-express --save* in your console in order to add it to your dependencies.

In the *app.js* file, or wherever you have your bootstrap configuration, you have to add the package and the documentation:

```
const swaggerUi = require('swagger-ui-express');
const openApiDocumentation = require('./openApiDocumentation');
```

appRequirements.js hosted with ❤ by **GitHub** **view raw**

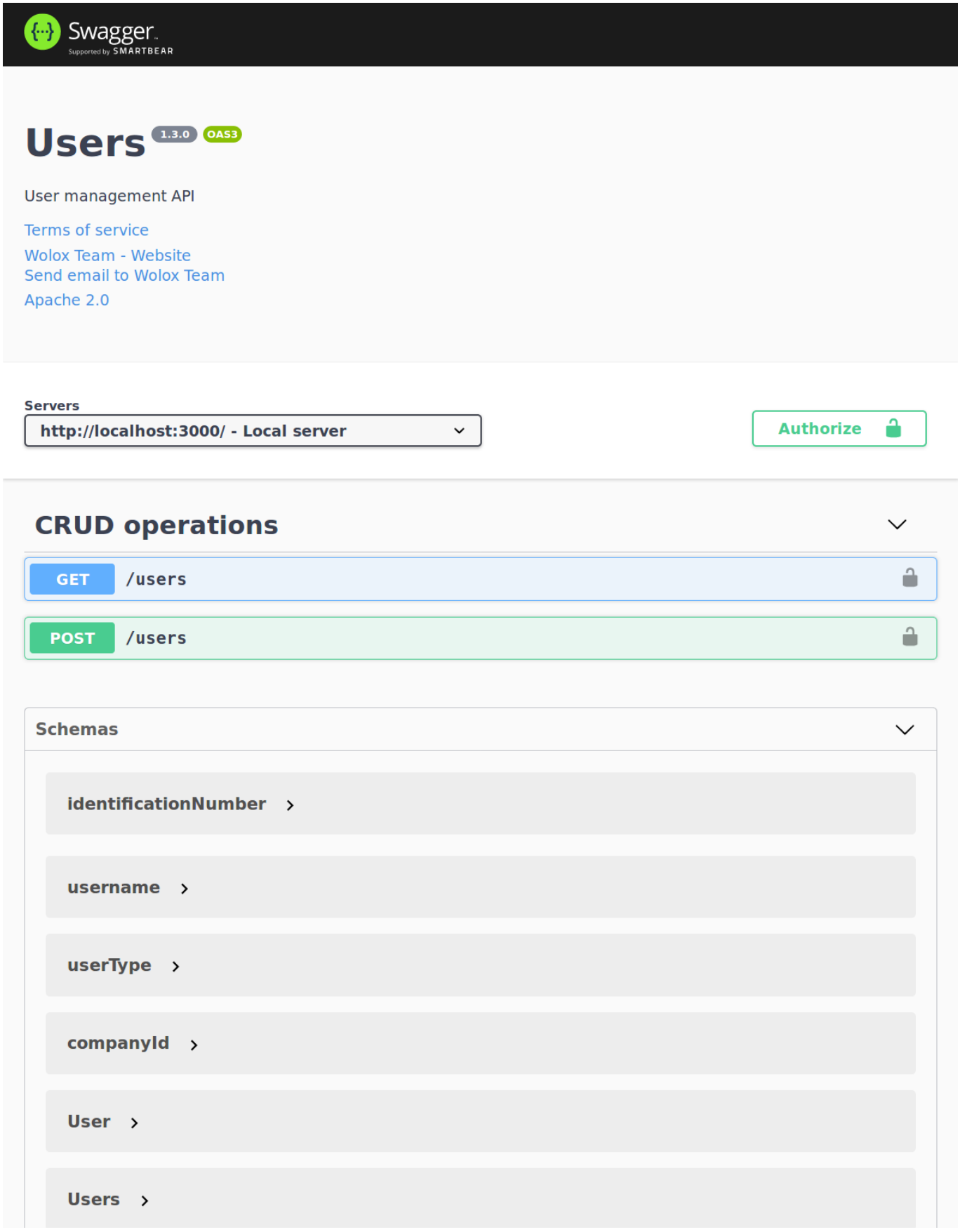
And the route where the documentation can be seen. I use */api-docs* in my example:

```
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(openApiDocumentation));
```

appDocumentationRoute.js hosted with ❤ by **GitHub** **view raw**

That is all!

Start the app and go to the route you have configured. You will see something like this:



Error ›

You can expand both the routes and the components to see more information.

If we look at the components, we will see all the information we have set in the documentation:

Schemas

identificationNumber integer
example: 1234
User identification number

username string
example: raparicio

userType string
default: regular
Enum:
[regular, admin, temporary]

companyId integer
example: 15
Company id where the user works

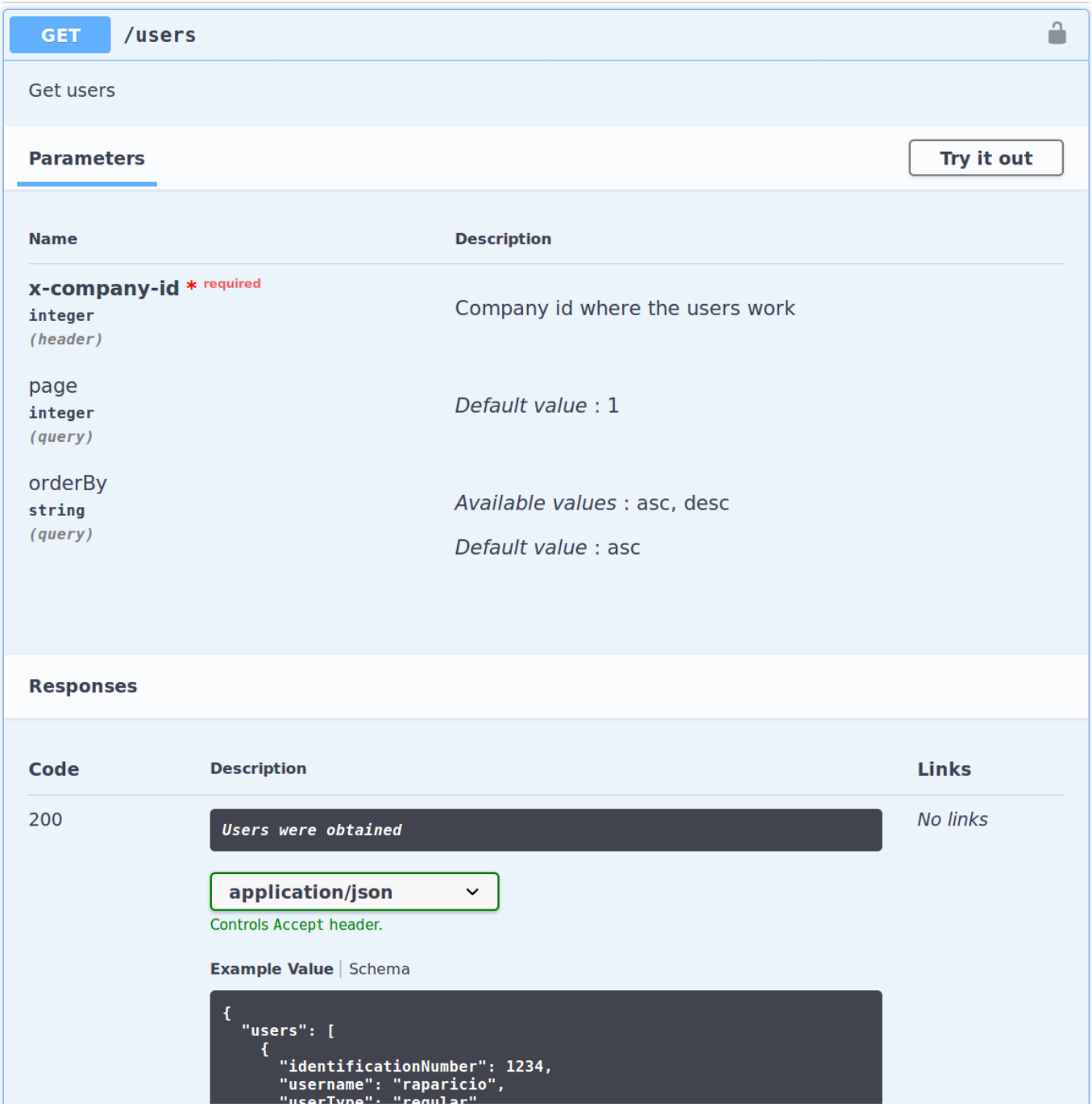
User {
identificationNumber identificationNumber integer
example: 1234
User identification number
username username string
example: raparicio
userType userType string
default: regular
Enum:
[regular, admin, temporary]
companyId companyId integer
example: 15
Company id where the user works
}

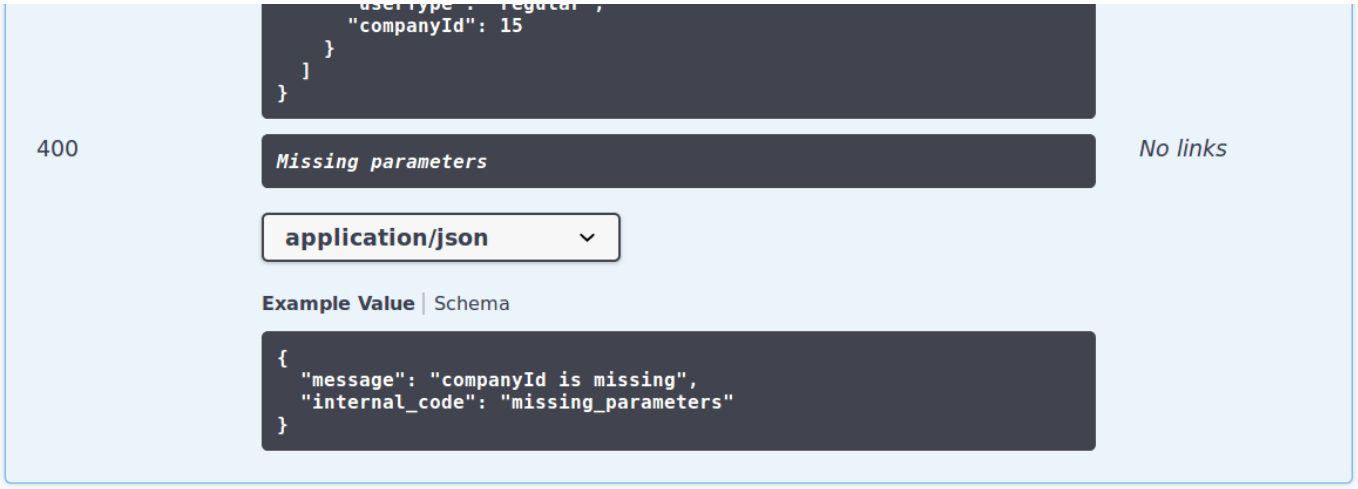
Users {
users [User › {...}]
}

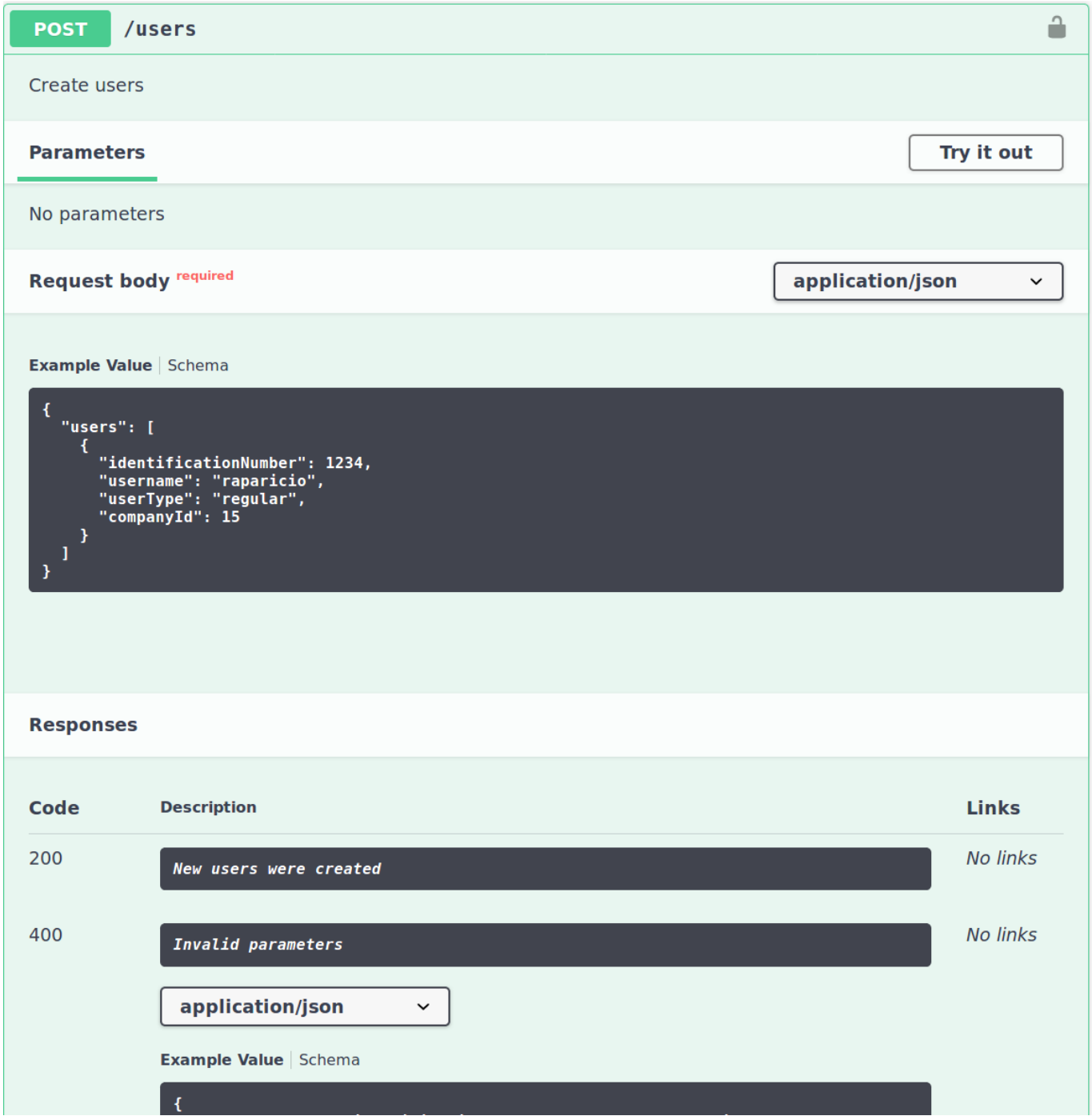
```
Error ⌄ {
   message            string
   internal_code      string
}
```

And then we have what, in my opinion, is the most interesting and powerful thing the interactive UI gives us.

Let's look at the endpoints:




"companyId": 15
}
]
}
400
Missing parameters
No links
application/json
Example Value | Schema
{
"message": "companyId is missing",
"internal_code": "missing_parameters"
}


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
New users were created
No links
400
Invalid parameters
No links
application/json
Example Value | Schema
{

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

It is amazing how each part we documented can be visualized.

But the cherry on top is the option to **try out** the endpoint. Like Postman, but on steroids! All required and not required parameters are already set, there are **examples** to easily see what you can use in the request, and responses are described to understand what we are getting from the server.

Before doing this, you should set the environment where you want to execute the requests. And if you have security like the apiKey in this example, there is an option to authorize the requests. Just with inserting one time the key value, all the requests you do will already have the key set. Both features are at the top of the page and can be easily modified.

Our results

The main advantage of using OpenApi 3 in our projects is **to save time**. Why? **Nobody has to explain how to use a documented API** because visual documentation speaks for itself.

My Quality Assurance teammate was very happy since he could perform all possible endpoints test cases without asking anything. **Everything was already set for him** so he only has to put some values and execute the endpoints.

The projects' Product Owners wanted to know about the API's progress and try the new features we had developed. It was impossible to show them the code as most of them do not know coding. On top of that, with Postman we could not explain clearly to them what the new features were about. Instead, after looking at the documentation UI, they were very satisfied with all the information they received. **We could establish a very positive way of showing them part of the progress.**

Last, but not least, other developers' teams found the documentation very helpful. When they need to make a request to a documented API, they do not waste time finding out how to do it properly, and also we do not have to explain anything to them.

As I described at the beginning, this project is very important for the company so we are frequently adding more developers. We find this documentation essential to avoid them a headache when reading all the code in the API's.

. . .

Conclusion

To sum up, you should concentrate your efforts on documenting your API well, in order to save a lot of time in the future. As you know, time is money.

If you have any questions, comments or suggestions, please let me know. I am more than willing to receive them. Here you can see the complete example used in the article.

Nodejs JavaScript Swagger Programming Software Engineering