API Requirements DRAFT (v0.1)

Basic Requirements

Should conform to REST specification.

Should return the right response according to the real situation. At least (200/201, 400/401/404, 500)

Should be segregated by the Read operations and Create/Update/Delete operations because of highly possible future optimizations.

Documentation

Each endpoint should have a basic description

Each endpoint should have ALL Possible response statuses according to designed behavior

Each endpoint should have ALL Possible configuration parameters with a short description

Each endpoint should have ALL Request / Response Entity description

in order to make easy interoperation between Frontend, Backend and other our and third-party services

Code Organization

Codebase reusing. Because we use JavaScript on FrontEnd and BackEnd we can reuse our code database

Documentation should be written in code by using corresponding annotations and decorators

...

@ApiOperation({ title: '...', description: '...' })

@ApiConsumes({ ... })

@ApiResponse({ status: ..., description: '...' })

...

Code segregation. Database models, Request DTO and Response DTO mustn't be connected in any way

DTO's should use internal validation by using the corresponding annotations and decorators:

...

@IsNotEmpty()

@IsString()

@IsDefined()

owner: string;

...

In some special cases, it is supposed to write custom validation decorators

...

@IsPhone()

phone: string;

@IsIndex(Locales.US)

phone: string;

...

By default, responses shouldn't expose more than 1 level nesting entities. If we need nesting entities, we should pass extra parameter somehow (for example: ?with=user, license)

A detailed canonical example will be presented in the separate document.

Test Coverage

Every endpoint should be shipped with the corresponding test case

The test should cover all the possible request/response/return status variants

...

request(app)

.post('/users')

.send({email, password})

.expect(201)

...

request(app)

.get('/users')

.send()

.expect(200)

...

request(app)

.post('/users')

.send(wrong\_req\_object)

.expect(400)

...

The test should cover all the corner cases like an empty value, invalid value, extra parameter, and missing parameter

The test should check the presence of all properties in the corresponding DTO's

The test should check all the properties have the type declared in the corresponding DTO

...

request(app)

.post('/users')

.send({email, password})

.expect(201)

.expect((res) => {

expect(res.header['x-auth']).toExist();

expect(res.body.\_id).toExist();

expect(res.body.\_id).toBeA('number');

expect(res.body.email).toBe(email);

})

…

A detailed canonical example will be presented in the separate document.

Security Requirements

All requests should be performed only in a secure way. It means each request should contain a specific header with some token given after the authorization process

All tokens should have a limited expiry time

The token should be given after a successful authentication and authorization stage

All requests without tokens, with wrong and expired tokens, should be rejected

All requests should be logged in the separate database for future analysis and development behavior rules.

List of fields to be logged:

endpoint path

request remote IP address

request date + time

request duration

request query

request response size

Extra Requirements

…

...