

APIs C# ASP.NET and NodeJS

The purpose of this tutorial is to understand how you can have two APIs, one in C# and another in NodeJS, linked together. We'll start by creating an API in C#, whose data will be stored in a SQLite database (section 1.1). Next, we will check how we can work with NodeJS and the Express framework (section 1.2) and how to implement a RESTful API using these technologies (section 1.3). We will also learn how to create a NodeJS API that communicates with the C# API (secção 1.4).

At the end, in the chapter 2, we will create an API in C# to work on data from a University (section 2.1) and an API in NodeJS to work on data from a university canteen, which will connect to the C# API to get data about students (section 2.2).

1 Tutorials

1.1 TodoItem API

This step consists of following the tutorial "Tutorial: Create a web API with ASP.NET Core"¹, whose objective is to create an API to access a list of tasks (*todo items*). This section does not include the entire tutorial, but only some parts where doubts may arise. For simplicity, the following titles correspond to the titles present in the tutorial.

1.1.1 Prerequisites

Install the necessary *software* to run the tutorial. Some important notes:

- Install .NET 6.0
- Pay attention to the .NET version selected in the tutorial (top left)

⚠ Using another version of .Net implies changing the versions of the packages to be installed later on

1.1.2 Create a web project

In the Visual Studio Code (VSCode) terminal, run some commands:

- Create the required folder structure for the `TodoApi` project:


```
dotnet new webapi -o TodoApi
```
- Change the current folder of the VSCode terminal to that of the project (enter the project):


```
cd TodoApi
```
- Add the `package Microsoft.EntityFrameworkCore.InMemory` to the project:


```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```
- Generate the necessary code for the project:


```
code -r ../TodoApi
```

¹<https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio-code>

1.1.3 Test the project

If this is the first time projects of this type are run on the computer, it is necessary to add the certificate. To do so, run the following command in the VSCode terminal:

```
dotnet dev-certs https --trust
```

Run the project by pressing the key combination **Ctrl** + **F5**.

If this key combination doesn't work:

1. On the left side of VSCode open "Run and Debug"
2. Click on the link that allows the creation of the file `launch.json`
3. Choose `.Net 6.0` and `.Net core`
4. Try the key combination again

Alternatively, if this key combination still doesn't work, run the command in the VSCode terminal:

```
dotnet run
```

If the command goes well, it should open a *browser* with a URL similar to <https://localhost:7143/> (instead of 7143 there may be another number, which corresponds to the port in the application is running). Navigate to <https://localhost:7143/swagger> to test the application.

1.1.4 Add a model class

1. In VSCode, add a `Models` folder.
2. Inside that folder, create a class `TodoItem.cs`.
3. Paste the code present in the tutorial in the file.

This code indicates that the entity `TodoItem` is characterized by:

- Id (of type `long`)
- Name (of type `string`)
- IsComplete (of type `boolean`)

⚠ Create the file with the correct name, including the extension

⚠ footnote-sizeAfter pasting the code into the file, save the file

⚠ To avoid having to remember to save the files, we can enable *Auto Save*: File > Auto Save)

1.1.5 Add a database context

In the `Models` folder, create a `TodoContext.cs` class. Paste the code present in the tutorial in the file.

This code defines the context of the API, its constructor, and even that the context contains a list of tasks:

```
public DbSet<TodoItem> TodoItems { get; set; } = null!;
```

In this code, the `!` indicates that the compiler should not issue a *warning* if the value is `null`².

²For more information about *null-forgiving*, consult <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-forgiving>

1.1.6 Scaffold a controller

- Add some *packages* to the project:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

- If this is the first time you are doing a project of this kind, install *Code Generator*: a tool that allows, from a model and a context, to create the API route controller³:

```
dotnet tool install -g dotnet-aspnet-codegenerator
```

- Execute *Code generator*, in order to create:

- The controller `TodoItemsController.cs` (argument: `-name TodoItemsController`)
- From the template `TodoItem.cs` (argument: `-m TodoItem`)
- E of the context `TodoContext.cs` (argument: `-dc TodoContext`)
- The controller is stored in the folder `Controllers` (argument: `-outDir Controllers`)

```
dotnet aspnet-codegenerator controller -name TodoItemsController
-async -api -m TodoItem -dc TodoContext -outDir Controllers
```

This command will create the `Controllers/TodoItemsController.cs` file, with several methods. Each method will be associated with a route according to Table 1.

Tabela 1: Routes created by *Code generator* in `TodoItemsController`.

Route	Note	Method	Description
GET: <code>api/TodoItems</code>	[HttpGet]	<code>Task<ActionResult<IEnumerable<TodoItem>>> GetTodoItems()</code>	Get list of <code>TodoItems</code>
GET: <code>api/TodoItems/5</code>	[HttpGet("id")]	<code>Task<ActionResult<TodoItem>> GetTodoItem(long id)</code>	Get <code>TodoItem</code> with <code>Id=5</code>
PUT: <code>api/TodoItems/5</code>	[HttpPut("id")]	<code>Task<ActionResult> PutTodoItem(long id, TodoItem todoItem)</code>	Replace <code>TodoItem</code> with <code>Id=5</code> with the one sent in the request body
POST: <code>api/TodoItems</code>	[HttpPost]	<code>Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)</code>	Create a <code>TodoItem</code>
DELETE: <code>api/TodoItems/5</code>	[HttpDelete("id")]	<code>Task<ActionResult> DeleteTodoItem(long id)</code>	Delete <code>TodoItem</code> with <code>Id=5</code>

1.1.7 Install http-repl

Instead, for a more user-friendly environment, you can install Postman⁴. When opening Postman, don't forget to disable *SSL certificate verification*.

³For more information about *Code generator*, see <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/tools/dotnet-aspnet-codegenerator?view=aspnetcore-6.0>

⁴Download: <https://www.getpostman.com/downloads/>

1.1.8 Test PostTodoItem

To test using Postman, just choose the method, introduce the URL and (if necessary) the body of the request, according to Table 2, in the order they appear.

Tabela 2: Body of the requests to be sent to the API and the responses obtained.

Route	Order body	Order body Answer
POST: api/TodoItems ⁵	{ "name": "walk dog", "isComplete": true }	{ "id": 1, "name": "walk dog", "isComplete": true }
POST: api/TodoItems ⁶	{ "name": "feed cat", "isComplete": false }	{ "id": 2, "name": "feed cat", "isComplete": false }
GET: api/TodoItems ⁷	<empty>	[{ "id": 1, "name": "walk dog", "isComplete": true }, { "id": 2, "name": "feed cat", "isComplete": false }]
GET: api/TodoItems/1 ⁸	<empty>	{ "id": 1, "name": "walk dog", "isComplete": true }

Continues on next page

⁵The "id" attribute is not provided in the request body. However, it appears in the response body. This happens because, as it is called Id, C# assumes that it is a primary key and will assign auto-incremented values starting at 1. For more information: <https://learn.microsoft.com/en-us/ef/core/modeling/keys?tabs=data-annotations>

⁶The same thing happens with the "id" attribute.

⁷Gets all entered todoitems - in this case 1 and 2

⁸Get the todoitem with id=1

Tabela 2 – Body of requests to be sent to the API and the responses obtained (cont.)

Route	Order body	Order body Answer
PUT: api/ToDoItems/1 ⁹	{ "id": 1, "name": "walk dog", "isComplete": false }	<empty>
GET: api/ToDoItems/1 ¹⁰	<empty>	{ "id": 1, "name": "walk dog", "isComplete": false }
DELETE: api/ToDoItems/1 ¹¹	<empty>	<empty>
GET: api/ToDoItems ¹²	<empty>	[{ "id": 2, "name": "feedcat", "isComplete": false }]

1.1.9 Routing and URL paths

Carefully read this part, including the recommended link "Attribute routing with Http[Verb] attributes."¹³.

1.1.10 Prevent over-posting

This section explains how a *Data Transfer Object* (DTO) works. The defined code indicates that the entity `ToDoItem` becomes characterized by:

- Id (of type `long`)
- Name (of type `string`)
- IsComplete (of type `boolean`)
- Secret (of type `string`)

⁹Will replace the `todoitem` with `id=1` with the one sent in the request body. In this case, it just changes one of the attributes. The url `id` must be the same as the request body

¹⁰Gets the `todoitem` with `id=1`. It is verified that the change made in the previous request was made.

¹¹Deletes the `todoitem` with `id=1`.

¹²Gets all entered `todoitems`. Verifies that the entire item with `id=1` has been deleted.

¹³<https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-6.0#verb>

In this case, the purpose of the DTO is to hide information that is not intended to be seen by the user (the Secret)¹⁴. For this, define the `TodoItemDTO`, characterized by:

- Id (of type `long`)
- Name (of type `string`)
- IsComplete (of type `boolean`)

1.1.11 Use a local SQLite DB

As per the link¹⁵. Run the command:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

In the `program.cs` file:

```
builder.Services.AddDbContext<TodoContext>(opt  
    =>opt.UseInMemoryDatabase("TodoList"));
```

passes to:

```
builder.Services.AddDbContext<TodoContext>(opt =>  
    opt.UseSqlite("Data Source = Todo.db"));
```

Run the commands:

```
dotnet tool install --global dotnet-ef  
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

¹⁴In Section 2.1 will be used for another purpose

¹⁵<https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>

1.2 Node.js Tutorial

Follow the tutorial¹⁶, taking into account the notes below.

1.2.1 Node.js - Express Framework

HelloWorldExample The code:

```
console.log("Example app listening at http://%s:%s", host, port)
```

show:

```
Example app listening at http://:::8081
```

why.

*":: is an IPv6 address in condensed form using the rule that a run of zeros can be replaced with ::."*¹⁷

File Upload (alternative 1) The code

```
app.use(multer({ dest: './tmp/' }));
```

gives error:

```
TypeError: app.use() requires a middleware function
```

To resolve, go to

```
app.use(multer({ dest: './tmp/' }).single('file'));
```

Then in server.js all occurrences of `files.file` have to pass only the file. Furthermore,

```
console.log(req.files.file.name);
console.log(req.files.file.path);
console.log(req.files.file.type);
var file = __dirname + "/" + req.files.file.name;
```

passes to

```
console.log("name: ", req.file.originalname);
console.log("path: ", req.file.path);
console.log("type: ", req.file.mimetype);
var file = __dirname + "/" + req.file.originalname;
```

and further down, inside the function

```
filename:req.files.file.name
```

passes to

```
filename:req.file.originalname
```

File Upload (alternative 2) Using¹⁸:

```
var multer = require('multer');
var upload = multer({ dest: './tmp' });
```

¹⁶<https://www.tutorialspoint.com/nodejs/index.htm>

¹⁷According to <https://superuser.com/questions/661188/what-is-in-the-local-address-of-netstat-output>

¹⁸According to <https://stackoverflow.com/questions/31656178/typeerror-app-use-requires-middleware-function>

1.2.2 Node.js - RESTful API

This API has only a list of users, where each user is characterized by:

- name
- password
- profession
- id

List Users Use Postman to test. According to what is in the tutorial code, the URL to be introduced in Postman will be:

```
http://127.0.0.1:8081/listUsers
```

Add User In the next Section, we'll learn how to do a POST by sending the content in the body of the request, instead of being *hard coded* in the code. The URL to enter in Postman will be:

```
http://127.0.0.1:8081/addUser
```

Show Detail The URL to enter in Postman will be, for example:

```
http://127.0.0.1:8081/1
```

to show the details of the user who has ID 1.

Delete User The URL to enter in Postman will be, for example:

```
http://127.0.0.1:8081/deleteUser
```

to delete the user user2.

1.3 Build a RESTful API Using Node and Express 4

1.3.1 Defining the Node Packages

Use version ~6.0.2 from mongoose in the `package.json` and run the command in the terminal:
`npm install.`

1.3.2 Using Express Router and Routes

Route Middleware At this point instead of using the suggested *sample* database, we are going to use Atlas MongoDB.

1. Create a free account at <https://www.mongodb.com/>
2. Create a free cluster, called *bears*, authenticating with *Username and password* and adding IP address
3. After the DB is created, click on *Connect* and choose *Connect your application*
4. Copy the *connection string* similar to:

```
mongodb+srv://<username>:<password>@bears.hunzqjs.mongodb.net/  
?retryWrites=true&w=majority
```

to use later on (replacing `<username>` and `<password>` by the values entered earlier

5. In the code file `server.js` replace link:

```
mongodb://node:node@novus.modulusmongo.net:27017/example
```

through the link of our BD

6. Continue the tutorial from the point "Testing the Middleware".

1.3.3 Creating the Basic Routes

Creating a Bear When using Postman to POST a new *bear*, we will now use the format `x-www-form-urlencoded` (in the C# API we used JSON). In "KEY" we introduce the value "name" (name of the field in the database) and in "VALUE" the value we want to assign (example "zé", the name in the new bear).

Getting All Bears You need to remove the ; from the end of the previous method, getting:

```
router.route('/bears')
  .post(function (req, res) {...
  })

  // get all the bears (accessed at GET http://localhost:8080/api/bears)
  .get(function (req, res) {
    ...
  });
```

That is, both .post like .get concern router.route('/bears'). The same will happen further down for another GET / PUT / DELETE.

The field "__v" that appears in the bears created concerns the version of the object (document) in the database.

1.3.4 Creating Routes for A Single Item

Getting a Single Bear The ID to be searched for is the one inserted (automatically) in the document. We can first do a normal GET to copy one of the Ids to use in this step.

1.4 Create a NodeJS API that communicates with the section's C# API [1.1](#)

The files referred to here can be found in a ZIP in Moodle.

1.4.1 App.js file

```
// BASE SETUP
// =====
var express = require('express'); // call express
var app = express(); // define our app using express
var bodyParser = require('body-parser');

// configure app to use bodyParser()
// this will let us get the data from a POST
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

var port = process.env.PORT || 8080; // set our port

// middleware
// =====
var mWare=require('./middleware');
app.use(mWare);

// API Routes
// =====
var taskRoutes=require('./routes/taskRoutes');
app.use('/api/tasks',taskRoutes);

// START THE SERVER
// =====
app.listen(port);
console.log('Magic happens on port ' + port);
```

1.4.2 Middleware.js file

```
// middleware to use for all requests

var express = require('express');
var router = express.Router();

router.use(function(req, res, next) {
  console.log(req.method + ' : ' + req.url)
  next();
});

module.exports = router;
```

1.4.3 Routese.js tasks file

```
var express = require('express'); // call express

var router = express.Router();

var Cli_API_TODOItems = require('node-rest-client').Client;

/* GET tasks: fetch all todoitems from the other API */
router.get('/', function (req, res, next) {
  var customer = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0; // Inhibits certificate verification
  client.get("https://localhost:7143/api/todoitems", function (data, response) {
    res.json(data); // send the data in the response
  });
});

/* GET task: fetch a todoitems from the other API (by id) */
router.get('/:id', function (req, res, next) {
  var customer = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0; // Inhibits certificate verification
  client.get("https://localhost:7143/api/todoitems/"
    + req.params.id, function (data, response) {
    res.json(data); // send the data in the response
  });
});

/* POST task: create a new task in the other API */
router.post("/", function (req, res, next) {
  var customer = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0; // Inhibits certificate verification
  var args = {
    data: { name: req.body.name, isComplete: req.body.isComplete },
    // we are assuming that we are going to introduce a json with the same
    // structure we used in the other api
    headers: { "Content-Type": "application/json" }
  };
  console.log(args);
  client.post("https://localhost:7143/api/todoitems/", args, function (data, response) {
    if (response.statusCode == 201) {
      if (err)
        res.send(err);
      res.json({ message: 'TodoItem successfully created' });
    } else {
      res.json({ message: 'An error occurred: ' + response.statusCode });
    }
  }).on('error', function (err) {
    console.log('An error occurred', err.request.options);
  });
});
```

```
/* PUT task: change a task in the other API */
router.put("/:id", function (req, res, next) {
  var customer = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0; // Inhibits certificate verification
  var args = {
    data: { id: req.body.id, name: req.body.name, isComplete: req.body.isComplete },
    // we are assuming that we are going to introduce a json with the same
    // structure we used in the other api
    headers: { "Content-Type": "application/json" }
  };
  console.log(args);
  client.put("https://localhost:7143/api/todoitems/"
    + req.params.id, args, function(data, response) {
    if (response.statusCode == 204) {
      res.json({ message: 'TodoItem successfully changed' });
    } else {
      res.json({ message: 'An error occurred: ' + response.statusCode });
    }
  }).on('error', function (err) {
    console.log('An error occurred', err.request.options);
  });
});

module.exports = router;
```

2 Exercise

2.1 University API

We start by creating a new API in a similar way to the one used for the `TodoItemsAPI` (Section 1.1.2¹⁹), running, in the VSCode terminal :

```
dotnet new webapi -o UniversityApi
cd UniversityApi
dotnet add package Microsoft.EntityFrameworkCore.InMemory
code -r ../UniversityApi
```

In the `Properties/launchSettings.json` file, change the line content:

```
"launchUrl": "swagger",
```

for

```
"launchUrl": "api/",
```

Then add the remaining packages needed:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

The purpose of this API is to store information about a University, composed of entities of different types:

- **Course:** characterized by an `Id` (long), an abbreviation (string) and a name (string)
- **Student:** characterized by an `Id` (long), a name (string), a course (Course) and a list of curricular units (`ICollection <Curricular Unit>`) to which you are enrolled
- **Curricular Unit:** characterized by an `id` (long), an acronym (string), a name (string), a course (Curso) and one year (int)
- **Note:** characterized by an `Id` (long), a value (int), the curricular unit to which it relates (`Curricular Unit`), as well as the student to which it is associated (`Student`).

It is necessary to create the folder `Models` to include the models (classes) associated with each of the entities, the respective DTOs (when necessary), and also the file `UniversidadeContext.cs`. As we implement models, to generate the respective *controller* (responsible for implementing the routes, referred to below), we will use the command:

```
dotnet-aspnet-codegenerator controller
-name XXXController -async -api
-m XXX -dc YYYContext -outDir Controllers
```

In the command, `XXX` and `YYY` will be replaced by the model and *controller* names, respectively. As mentioned, the API will implement several routes, associated with each of the models/*controllers*. This project will be carried out in several phases, associated with the models and explained in the following sections.

¹⁹Carefully analyze the differences between these commands and those executed for the first API

? footnote-sizeStart by outlining the new API

⚠ The API will have just one context, which handles all entities

2.1.1 Course

Models/Course.cs A Course is characterized by:

- Id (of type long)
- Acronym (of the type string), the abbreviation of the Course
- Name (of type string), the name of the Course

UniversityContext.cs This code will define the context of the API, its constructor, and even that the context contains a list of courses. It is also necessary to change the `Program.cs` file, adapting the changes made in the case of `TodoItemsAPI`.

CourseController.cs In order to generate the *controller* for the course, execute the command:

```
dotnet-aspnet-codegenerator controller
-name CourseController -async -api
-m Course -dc UniversityContext -outDir Controllers
```

For the course, we intend to implement the routes presented in the Table 3.

? Fill in the table with the information in this

Tabela 3: University - Entity Course API routes.

Route	Note	Method	Description
GET: api/cursos			Allows you to get all existing courses in the repository
GET: api/cursos/1			Allows to get a course by relational id
GET: api/cursos/LEI			Allows to obtain a course by acronym
POST: api/cursos			Allows the creation of a new course
PUT: api/cursos/1			Allows editing a course
DELETE: api/cursos/1			Allows the deletion of a course

The routes described in the table are similar to those implemented in the previous API. The biggest difference in this case is that we have the GET: `api/cursos/LEI` route, which:

- Has a "similar" route to GET: `api/cursos/1`²⁰
- Will get the course from the acronym, which not is the relational id²¹

²⁰For more information about *routing* in .Net:

- <https://learn.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/create-a-rest-api-with-attribute-routing>
- <https://devblogs.microsoft.com/dotnet/attribute-routing-in-asp-net-mvc-5/#route-constraints>

If you change the name of the method that does the GET by Id, you also need to change the last line of the method that controls the POST.

²¹See previous footnote links.

The Table 4 contains examples of requests to be made to the API, in the order in which they should be made.

? Fill in the table with the missing information

Tabela 4: Body of the requests to be sent to the API and the responses obtained - Course Entity.

Route	Order body	Order body Answer
POST: api/courses	{ "acronym": "LAW", "name": "L E I" }	?
POST: api/courses	{ "acronym": "LEGI", "name": "Lic. Eng. Management Ind." }	?
POST: api/courses	?	{ "id": 3, "acronym": "LSIG", "name": "Lic. System Inf. Management." }
PUT: api/courses/1	{ "id": 1, "acronym": "LAW", "name": "Lic. Eng. Informatics" }	?
GET: api/courses	?	?
GET: api/courses/1	?	?
GET: api/courses/LES	?	?

2.1.2 Student

Student.cs A Student is characterized by:

- Id (of type long)
- Name (of type string), the name of the Student
- Course (of type Course), the Course what the Student is subscribed
- Balance (of type double, the balance of Student (initial balance: €100)
- Email (of type string), the email of Student

Update UniversityContext.cs The API context will also have to include a list of students.

StudentsController.cs Create the *controller* based on the Student model.

AlunoDTO.cs We do not want that, when inserting a student, it is mandatory to enter all the information about the course (Id, Acronym, Name); or that, when viewing a student, all the information of his course appears. It would be easier for the user to be allowed to enter or view only, for example, the abbreviation of the respective course. Thus, we will have to create a DTO for a student, where a StudentDTO is characterized by:

- Id (of type long)
- Name (of type string), the name of the Student
- SiglaCurso (of type string), the abbreviation of Course what the Student is subscribed
- Balance (of type double, the balance of Student (initial balance: €100)
- Email (of type string), the email of Student

It will still be necessary to update *controller* to use AlunoDTO instead of Student²². In the end, it will be possible to use the routes presented in the Table 5.


 Fill in the table with the missing information

Tabela 5: University - Entity Student API routes.

Route	Note	Method	Description
GET: api/students			Allows to get all existing students in the repository
GET: api/students/1			Allows to get a student by relational id
POST: api/students			Allows the creation of a new student
PUT: api/students/1			Allow editing of a student
DELETE: api/students/1			Allows the deletion of a student

²²See documentation at <https://learn.microsoft.com/en-us/ef/ef6/querying/related-data> and <https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-4>

The Table 6 contains examples of requests to be made to the API, in the order in which they should be made.

? Fill in the table with the missing information

Tabela 6: Body of the requests to be sent to the API and the responses obtained - Student Entity.

Route	Order body	Order body Answer
POST: api/students	{ "name": "Bert", "acronym": "LAW" }	?
POST: api/students	{ "name": "Anna", "siglcurso": "LEGI" }	{ "id": 2, "name": "Anna", "siglcurso": "LEGI" }
POST: api/students	?	{ "id": 3, "name": "Carlos", "siglcurso": "LSIG" }
PUT: api/students/1	{ "id": 1, "name": "Albert", "acronym": "LAW" }	?
GET: api/students	?	?
GET: api/students/1	?	?

2.1.3 Curricular Unit

Curricular Unit.cs A Curricular Unit is characterized by:

- Id (of type long)
- Acronym (of the type string), the acronym of the Curricular Unit
- Name (of type string), the name of the Curricular Unit
- Course (of type Course), the Course to which Curricular Unit
- Year (of type int), the year of the Course in which the Curricular Unit

Update UniversityContext.cs The context of the API will also have to include a list of curricular units.

UCsController.cs Create the *controller* based on the model Curricular Unit.

UC_DTO.cs For the same reason as for Student, we need to create a DTO for Curricular Unit, which is characterized by:

- Id (of type long)
- Acronym(of the string type), the acronym of the Curricular Unit
- Name (of type string), the name of the Curricular Unit
- SiglaCurso (of type string), the abbreviation of Course to which Curricular Unit
- Year (of type int), the year of the Course in which the Curricular Unit

It will still be necessary to update the *controller* to use the UC_DTO instead of Curricular Unit. In the end, it will be possible to use the routes presented in the Table 7.

? Fill in the table with the missing information

Tabela 7: University API Routes - Curricular Unit Entity.

Route	Note	Method	Description
GET: api/ucs			Allows you to get all existing course units in the repository
GET: api/ucs/1			Allows to obtain the curricular unit by relational id
GET: api/ucs/SINF2			Allows to obtain the curricular unit by the initials
POST: api/ucs			Allows creation of a new uc
PUT: api/ucs/1			Allows editing of a uc
DELETE: api/ucs/1			Allows deletion of a uc

The Table 8 contains examples of requests to be made to the API, in the order in which they should be made.

? Fill in the table with the missing information

Tabela 8: Body of the requests to be sent to the API and the responses obtained - Curricular Unit Entity.

Route	Order body	Order body Answer
POST: api/ucs	{ "acronym": "LTW", "Name": "Web Technologies", "acronym": "LAW", "year": 2 }	?
POST: api/ucs	{ "acronym": "ALPROGI", "Name": "Something. e Prog LEI", "acronym": "LAW", "year": 1 }	{ "id": 2, "acronym": "ALPROGI", "Name": "Something. and Prog. LAW", "acronym": "LAW", "year": 1 }
POST: api/ucs	{ "acronym": "ALPROGII", "Name": "Something. and Prog. LEGI", "acronymcourse": "LEGI", "year": 1 }	{ "id": 3, "acronym": "ALPROGII", "Name": "Something. and Prog. LEGI", "acronymcourse": "LEGI", "year": 1 }
POST: api/ucs	?	{ "id": 4, "acronym": "POO", "Name": "Prog. Or. Obj.", "acronym": "LAW", "year": 1 }
PUT: api/ucs/1	{ "id": 1, "acronym": "LTW", "Name": "Web Tech Lab", "acronym": "LAW", "year": 2 }	?
GET: api/ucs	?	?

Continues on next page

Tabela 8 – Body of the requests to be sent to the API and the responses obtained - Entity UnidadeCurricular (cont.)

Route	Order body	Order body Answer
GET: api/ucs/1	?	?
GET: api/ucs/LTW	?	?

2.1.4 Note

Note.cs A Note is characterized by:

- Id (of type long)
- Value (of type double), the value of the Note obtained, from 0.0 to 20.0
- CurricularUnit (of the type CurricularUnit), the CurricularUnit the one concerning Note
- Student (of type Student), the Student to which Note in this Curricular Unit

Update UniversityContext.cs The API context will also have to include a list of notes.

NotesController.cs Create the *controller* based on the Note template.

Note.DTO.cs For the same reason as for the Student and Curricular Unit, we need to create a DTO for Note, which is characterized by:

- Id (of type long)
- Value (of type double), the value of the Note obtained, from 0.0 to 20.0
- SiglaUC (like string), the acronym of the Curricular Unit the one concerning Note
- StudentName (of type string), the name of Student to which Note in this Curricular Unit

It will still be necessary to update the *controller* to use the Nota.DTO instead of Note. In the end, it will be possible to use the routes presented in the Table 9.

Fill in the table with the missing information

Tabela 9: University API Routes - Note Entity.

Route	Note	Method	Description
GET: api/notes			Allows you to get all existing notes in the repository
GET: api/grades/1			Allows to get a student's grades by student id
GET: api/notes/ALPROGII			Allows to obtain the grades of a curricular unit by the abbreviation of the curricular unit
POST: api/notes			Allows the creation of a new uc
PUT: api/notes/1			Allows editing of a uc
DELETE: api/notes/1			Allows deletion of a uc

The Table 10 contains examples of requests to be made to the API, in the order in which they should be made.

Fill in the table with the missing information

Tabela 10: Body of the requests to be sent to the API and the responses obtained - Entity Note.

Route	Order body	Order body Answer
POST: api/notes	{ "value":10.0, "studentname":"Alberto", "siglauc":"ALPROGI" }	?
POST: api/notes	{ "value":15.0, "studentname":"Carlos", "siglauc":"ALPROGII" }	{ "id":2, "value":15.0, "studentname":"Carlos", "siglauc":"ALPROGII" }
POST: api/notes	{ "value":11.0, "studentname":"Ana", "siglauc":"POO" }	{ "id":3, "value":11.0, "studentname":"Ana", "siglauc":"POO" }
PUT: api/notes/1	{ "id": 1, "value": 12, "StudentName": "Alberto", "acronymUC": "ALPROGI" }	?
GET: api/notes	?	?
GET: api/notes/1	?	?
GET: api/ucs/ALPROGI	?	?

2.1.5 Migrating to a SQLite DB

As mentioned earlier in the section [1.1.11](#).

2.1.6 Possible improvements

Possible improvements that could be added to the API (among others):

- Implement the route GET: `api/aluno/EI`, which obtains all students enrolled in the degree whose acronym is given
- Add a list of course units to the student
- When enrolling a student in a UC, validate that the UC is the course in which the student is enrolled
- When inserting a grade, validate that the student is enrolled in this UC
- ...

2.2 Cantina API

In order to store information about a Cantina, it is necessary to build a NodeJS API:

- Using express-generator²³
- Using a MongoDB database, hosted on Atlas mongoDB²⁴
- That communicates with the C# API created earlier, as mentioned in the section 2.2.3

Below is the information that must be saved, along with the routes that must be implemented in the API.

2.2.1 Dish of the day

A dish of the day is characterized by:

- `id` (of type `Schema.Types.ObjectId`)
- `dish_name` (of type `String`), the name of the dish
- `dia_prato` (of type `String`), the day to which the dish corresponds

The routes to be implemented, associated with the entity *Prato do dia*, are shown in Table 11.

Tabela 11: Cantina API routes - Prato do dia Entity.

Route	HTTP Verb	Description
<code>/api/prato</code>	POST	create a <code>PratoDoDia</code>
<code>/api/prato</code>	GET	list all <code>PratoDoDia</code>
<code>/api/prato/28412548h2123</code>	GET	list a <code>PratoDoDia</code>
<code>/api/prato/28412548h2123</code>	PUT	edit a <code>PratoDoDia</code>
<code>/api/prato/28412548h2123</code>	DELETE	remove a <code>PratoDoDia</code>

²³<https://developers.sap.com/tutorials/basic-nodejs-application-create.html>

²⁴<https://www.mongodb.com/>

2.2.2 Menu of the week

A menu of the week is characterized by:

- `id` (of type `Schema.Types.ObjectId`)
- `data` (of type `Date`), the date to which the menu corresponds, with value `default Date.now`
- `dishlist` (a list of objects of type `Dish`), the list of dishes for that week

The routes to be implemented, associated with the *Menu of the week* entity, are shown in Table 12.

Tabela 12: Cantina API routes - Entity Menu of the week.

Route	HTTP Verb	Description
/api/menu	POST	create a menu
/api/menu	GET	list all menus
/api/menu/28412548h2123	GET	list a menu
/api/menu/28412548h2123	DELETE	remove a menu

2.2.3 Reservation

A reservation is characterized by:

- `id` (of type `Schema.Types.ObjectId`)
- `data` (of type `Date`), the date to which the reservation corresponds, with value `default Date.now`
- `pratoReservado` (of type `Prato`), the reserved dish
- `student` (an object that stores `Num_student`, `Name_student` and `Email_student`)

The routes to be implemented, associated with the entity *Reserva*, are shown in Table 13.

Tabela 13: Cantina API routes - Reservation Entity.

Route	HTTP Verb	Description
/api/reservation	POST	create a reservation
/api/reservation	GET	list reservations
/api/reservas/1980980	GET	list a student's reservations
/api/reservation/1980980	DELETE	remove a student's reservations

When **creating a reservation**, it is necessary to call the API made in C#, using `node-rest-client`²⁵, to obtain student data and update your balance (assume the booking price is €4.00). A new route will have to be created capable of updating the student balance when booking a meal.

²⁵<https://www.npmjs.com/package/node-rest-client>