

APIs C# ASP.NET e NodeJS

O objetivo deste tutorial é perceber como se pode ter dumas APIs, uma em C# e outra em NodeJS, ligadas entre si. Iremos começar por criar uma API em C#, cujos dados serão guardados numa BD SQLite (secção 1.1). De seguida, iremos verificar como podemos trabalhar com o NodeJS e framework Express (secção 1.2) e como se implementa uma API RESTful utilizando essas tecnologias (secção 1.3). Iremos também aprender como criar uma API NodeJS que comunica com a API C# (secção 1.4).

No final, no capítulo 2, iremos criar uma API em C# para trabalhar sobre os dados de uma Universidade (secção 2.1) e uma API em NodeJS para trabalhar sobre os dados de uma cantina de uma universidade, que se ligará à API C# para obter dados sobre os alunos (secção 2.2).

1 Tutoriais


1.1 API TodoItem

Este passo consiste em seguir o tutorial "Tutorial: Create a web API with ASP.NET Core"¹, cujo objetivo é criar uma API para acesso a uma lista de tarefas (*todo items*). Esta secção não inclui a totalidade do tutorial, mas apenas algumas partes em que possam surgir dúvidas. Para simplificação, os títulos seguintes correspondem aos títulos presentes no tutorial.

1.1.1 Prerequisites

Instalar o *software* necessário para a execução do tutorial. Algumas notas importantes:

- Instalar o .NET 6.0
- Atenção à versão do .NET selecionada no tutorial (topo, lado esquerdo)

 A utilização de uma outra versão do .Net implica alteração das versões dos packages a instalar mais à frente

1.1.2 Create a web project

No terminal do Visual Studio Code (VSCode), executar alguns comandos:

- Criar a estrutura de pastas necessária para o projeto TodoApi:

```
dotnet new webapi -o TodoApi
```

- Mudar a pasta atual do terminal do VSCode para a do projeto (entrar no projeto):

```
cd TodoApi
```

- Adicionar o *package* `Microsoft.EntityFrameworkCore.InMemory` ao projeto:

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

- Gerar o código necessário para o projeto:

```
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

Caso seja a primeira vez que se executa projetos deste tipo no computador, é necessário adicionar o certificado. Para isso, executar no terminal do VSCode o seguinte comando:

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

Executar o projeto premindo a combinação de teclas **Ctrl** + **F5**.

Caso esta combinação de teclas não funcione:

1. Do lado esquerdo do VSCode abrir "Run and Debug"
2. Clicar no link que permite a criação do ficheiro `launch.json`
3. Escolher `.Net 6.0 and .Net core`
4. Voltar a experimentar a combinação de teclas

Em alternativa, caso esta combinação de teclas continue a não funcionar, executar no terminal do VSCode o comando:

```
dotnet run
```

Se o comando correr bem, deverá abrir um *browser* com um URL semelhante a <https://localhost:7143/> (em vez de 7143 poderá estar outro número, que corresponde à porta em que a aplicação está a correr). Navegar até <https://localhost:7143/swagger> para testar a aplicação.

1.1.4 Add a model class

1. No VSCode, adicionar uma pasta `Models`.
2. Dentro dessa pasta, criar uma classe `TodoItem.cs`.
3. Colar no ficheiro o código presente no tutorial.

Este código indica que a entidade `TodoItem` é caracterizado por:

- Id (do tipo `long`)
- Name (do tipo `string`)
- IsComplete (do tipo `boolean`)

⚠ Criar o ficheiro com o nome correto, incluindo a extensão

⚠ Depois de colar o código no ficheiro, guardar o ficheiro

⚠ Para evitar ter de lembrar de guardar os ficheiros, podemos ativar o *Auto Save*: File > Auto Save)

1.1.5 Add a database context

Na pasta `Models`, criar uma classe `TodoContext.cs`. Colar no ficheiro o código presente no tutorial.

Este código define o contexto da API, o seu construtor, e ainda que o contexto contém uma lista de tarefas:

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

Neste código, o `!` indica que o compilado não deve emitir um *warning* caso o valor seja nulo².

²Para mais informação sobre o *null-forgiving*, consultar <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-forgiving>

1.1.6 Scaffold a controller

- Adicionar alguns *packages* ao projeto:

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

- Caso seja a primeira vez que se faz um projeto deste género, instalar o *Code Generator*: ferramenta que permite, a partir de um modelo e de um contexto, criar o controlador das rotas da API³:

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

- Executar o *Code generator*, de forma a criar:

- O controlador `TodoItemsController.cs` (argumento: `-name TodoItemsController`)
- A partir do modelo `TodoItem.cs` (argumento: `-m TodoItem`)
- E do contexto `TodoContext.cs` (argumento: `-dc TodoContext`)
- O controlador fica guardado na pasta `Controllers` (argumento: `-outDir Controllers`)

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

Este comando vai criar o ficheiro `Controllers/TodoItemsController.cs`, com vários métodos. Cada método será associado a uma rota de acordo com a Tabela 1.

Tabela 1: Rotas criadas pelo *Code generator* no `TodoItemsController`.

Rota	Anotação	Método	Descrição
GET: <code>api/TodoItems</code>	<code>[HttpGet]</code>	<code>Task<ActionResult<IEnumerable<TodoItem>>> GetTodoItems()</code>	Obter a lista dos <code>TodoItems</code>
GET: <code>api/TodoItems/5</code>	<code>[HttpGet("id")]</code>	<code>Task<ActionResult<TodoItem>> GetTodoItem(long id)</code>	Obtenção do <code>TodoItem</code> com <code>Id=5</code>
PUT: <code>api/TodoItems/5</code>	<code>[HttpPut("id")]</code>	<code>Task<ActionResult> PutTodoItem(long id, TodoItem todoItem)</code>	Substituir o <code>TodoItem</code> com <code>Id=5</code> pelo enviado no corpo do request
POST: <code>api/TodoItems</code>	<code>[HttpPost]</code>	<code>Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)</code>	Criar um <code>TodoItem</code>
DELETE: <code>api/TodoItems/5</code>	<code>[HttpDelete("id")]</code>	<code>Task<ActionResult> DeleteTodoItem(long id)</code>	Eliminar o <code>TodoItem</code> com <code>Id=5</code>

1.1.7 Install http-repl

Em vez disto, para um ambiente mais user-friendly, é possível instalar o Postman⁴. Ao abrir o Postman, não esquecer de desativar a *SSL certificate verification*.

³Para mais informação sobre o *Code generator*, consultar <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

Para testar usando o Postman, basta escolher o método, introduzir o URL e (caso necessário) o corpo do pedido, de acordo com a Tabela 2, pela ordem que aparecem.

Tabela 2: Corpo dos pedidos a enviar à API e das respostas obtidas.

Rota	Corpo do Pedido	Corpo da Resposta
POST: api/TodoItems ⁵	<pre>{ "name": "walk dog", "isComplete": true }</pre>	<pre>{ "id": 1, "name": "walk dog", "isComplete": true }</pre>
POST: api/TodoItems ⁶	<pre>{ "name": "feed cat", "isComplete": false }</pre>	<pre>{ "id": 2, "name": "feed cat", "isComplete": false }</pre>
GET: api/TodoItems ⁷	<vazio>	<pre>[{ "id": 1, "name": "walk dog", "isComplete": true }, { "id": 2, "name": "feed cat", "isComplete": false }]</pre>
GET: api/TodoItems/1 ⁸	<vazio>	<pre>{ "id": 1, "name": "walk dog", "isComplete": true }</pre>

Continua na próxima página

⁵O atributo "id" não é fornecido no corpo do pedido. No entanto, aparece no corpo da resposta. Isto acontece porque, por se chamar Id, o C# assume que é uma chave primária e vai atribuir-lhe valores iniciados em 1, auto-incrementados. Para mais informação: <https://learn.microsoft.com/en-us/ef/core/modeling/keys?tabs=data-annotations>

⁶Acontece o mesmo com o atributo "id".

⁷Obtém todos os todoitems inseridos - neste caso, o 1 e o 2

⁸Obtém o todoitem com id=1

Tabela 2 – Corpo dos pedidos a enviar à API e das respostas obtidas (cont.)

Rota	Corpo do Pedido	Corpo da Resposta
PUT: api/ToDoItems/1 ⁹	{ "id": 1, "name": "walk dog", "isComplete": false }	<vazio>
GET: api/ToDoItems/1 ¹⁰	<vazio>	{ "id": 1, "name": "walk dog", "isComplete": false }
DELETE: api/ToDoItems/1 ¹¹	<vazio>	<vazio>
GET: api/ToDoItems ¹²	<vazio>	[{ "id": 2, "name": "feed cat", "isComplete": false }]

1.1.9 Routing and URL paths

Ler atentamente esta parte, incluindo o link recomendado "Attribute routing with Http[Verb] attributes."¹³.

1.1.10 Prevent over-posting

Esta secção explica o funcionamento de um *Data Transfer Object* (DTO). O código definido indica que a entidade `TodoItem` passa a ser caracterizado por:

- Id (do tipo `long`)
- Name (do tipo `string`)
- IsComplete (do tipo `boolean`)
- Secret (do tipo `string`)

⁹Vai substituir o `todoitem` com `id=1` pelo enviado no corpo do pedido. Neste caso, apenas altera um dos atributos. O `id` do URL tem de ser o mesmo do corpo do pedido

¹⁰Obtém o `todoitem` com `id=1`. Verifica-se que a alteração efetuada no pedido anterior foi efetuada.

¹¹Elimina o `todoitem` com `id=1`.

¹²Obtém todos os `todoitems` inseridos. Verifica-se que o `todoitem` com `id=1` foi eliminado.

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

Neste caso, o objetivo do DTO é esconder informação que não se pretende que seja visualizada pelo utilizador (o Secret)¹⁴. Para isso, define-se o `TodoItemDTO`, caracterizado por:

- Id (do tipo `long`)
- Name (do tipo `string`)
- IsComplete (do tipo `boolean`)

1.1.11 Utilizar uma BD local SQLite

De acordo com o link¹⁵. Executar o comando:

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

No ficheiro `program.cs`:

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

passa a:

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

Executar os comandos:

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

¹⁴Na Secção 2.1 irá ser utilizado com outro propósito

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

1.2 Node.js Tutorial

Seguir o tutorial¹⁶, tendo em conta as notas apresentadas a seguir.

1.2.1 Node.js - Express Framework

Hello world Example O código:

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

mostra:

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

porque.

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

File Upload (alternativa 1) O código

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

dá erro:

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

Para resolver passa a

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

Depois, no server.js todas as ocorrências de `files.file` têm de passar só a `file`. Para além disso,

```
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;
```

passa a

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

e mais abaixo, dentro da função

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

passa a

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

File Upload (alternativa 2) Utilizar¹⁸:

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

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

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

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

1.2.2 Node.js - RESTful API

Esta API tem apenas uma lista de users, em que cada user é caracterizado por:

- name
- password
- profession
- id

List Users Usar o Postman para testar. De acordo com o que está no código do tutorial, o URL a introduzir no Postman será:

`http://127.0.0.1:8081/listUsers`

Add User Na Secção seguinte vamos aprender a fazer um POST enviando o conteúdo no corpo do pedido, em vez de estar *hard coded* no código. O URL a introduzir no Postman será:

`http://127.0.0.1:8081/addUser`

Show Detail O URL a introduzir no Postman será, por exemplo:

`http://127.0.0.1:8081/1`

para mostrar os detalhes do utilizador que tem ID 1.

Delete User O URL a introduzir no Postman será, por exemplo:

`http://127.0.0.1:8081/deleteUser`

para eliminar o utilizador user2.

1.3 Build a RESTful API Using Node and Express 4

1.3.1 Defining the Node Packages

Usar a versão ~6.0.2 do mongoose no ficheiro `package.json` e executar, no terminal, o comando:
`npm install.`

1.3.2 Using Express Router and Routes

Route Middleware Neste ponto em vez de usar a base de dados *sample* sugerida, vamos utilizar o Atlas MongoDB.

1. Criar uma conta grátis em <https://www.mongodb.com/>
2. Criar um cluster grátis, chamado *bears*, com autenticação *Username and password* e adicionando o endereço IP
3. Depois de a BD ser criada, clicar em *Connect* e escolher *Connect your application*
4. Copiar o *connection string* semelhante a:

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

para depois utilizar mais à frente (substituindo `<username>` e `<password>` pelos valores introduzidos anteriormente

5. No código do ficheiro `server.js` substituir o link:

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

pelo link da nossa BD

6. Continuar o tutorial a partir do ponto "Testing the Middleware".

1.3.3 Creating the Basic Routes

Creating a Bear Ao utilizar o Postman para criar (POST) um novo *bear*, agora vamos usar o formato `x-www-form-urlencoded` (na API C# usávamos JSON). Introduzimos em "KEY" o valor "name" (nome do campo na base de dados) e em "VALUE" o valor que queremos atribuir (exemplo "zé", o nome do novo urso).

Getting All Bears É necessário remover o ; do final do método anterior, ficando:

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

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

Ou seja, tanto o `.post` como o `.get` dizem respeito ao `router.route('/bears')`. O mesmo vai acontecer mais abaixo para outro GET / PUT / DELETE.

O campo `"__v"` que aparece nos ursos criados diz respeito à versão do objeto (documento) na base de dados.

1.3.4 Creating Routes for A Single Item

Getting a Single Bear O Id pelo qual se vai procurar é o inserido (automaticamente) no documento. Podemos primeiro fazer um GET normal para copiar um dos Ids para utilizar neste passo.

1.4 Criar uma API NodeJS que comunica com a API C# da secção 1.1

Os ficheiros aqui referidos encontram-se num ZIP no Moodle.

1.4.1 Ficheiro app.js

```
// 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 tarefasRoutes=require('./routes/tarefasRoutes');
app.use('/api/tarefas',tarefasRoutes);

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

1.4.2 Ficheiro middleware.js

```
// 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 Ficheiro tarefasRoutes.js

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

var router = express.Router();

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

/* GET tarefas: vai buscar todos os todoitems à outra API */
router.get('/', function (req, res, next) {
  var cliente = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0;    // Inibe a verificação dos certificados
  cliente.get("https://localhost:7143/api/todoitems", function (dados, response) {
    res.json(dados); // manda os dados na resposta
  });
});

/* GET tarefa: vai buscar um todoitems à outra API (pelo id) */
router.get('/:id', function (req, res, next) {
  var cliente = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0;    // Inibe a verificação dos certificados
  cliente.get("https://localhost:7143/api/todoitems/"
    + req.params.id, function (dados, response) {
    res.json(dados); // manda os dados na resposta
  });
});

/* POST tarefa: cria uma nova tarefa na outra API */
router.post("/", function (req, res, next) {
  var cliente = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0;    // Inibe a verificação dos certificados
  var args = {
    data: { name: req.body.name, isComplete: req.body.isComplete },
    // estamos a assumir que vamos introduzir um json com a mesma
    // estrutura que usavamos na outra api
    headers: { "Content-Type": "application/json" }
  };
  console.log(args);
  cliente.post("https://localhost:7143/api/todoitems/", args, function (dados, response) {
    if (response.statusCode == 201) {
      if (err)
        res.send(err);
      res.json({ message: 'TodoItem criado com sucesso' });
    } else {
      res.json({ mensagem: 'Ocorreu um erro: ' + response.statusCode });
    }
  }).on('error', function (err) {
    console.log('Ocorreu um erro', err.request.options);
  });
});
```

```
/* PUT tarefa: altera uma tarefa na outra API */
router.put("/:id", function (req, res, next) {
  var cliente = new Cli_API_TODOItems();
  process.env['NODE_TLS_REJECT_UNAUTHORIZED'] = 0; // Inibe a verificação dos certificados
  var args = {
    data: { id: req.body.id, name: req.body.name, isComplete: req.body.isComplete },
    // estamos a assumir que vamos introduzir um json com a mesma
    // estrutura que usavamos na outra api
    headers: { "Content-Type": "application/json" }
  };
  console.log(args);
  cliente.put("https://localhost:7143/api/todoitems/"
    + req.params.id, args, function (dados, response) {
    if (response.statusCode == 204) {
      res.json({ message: 'TodoItem alterado com sucesso' });
    } else {
      res.json({ mensagem: 'Ocorreu um erro: ' + response.statusCode });
    }
  }).on('error', function (err) {
    console.log('Ocorreu um erro', err.request.options);
  });
});

module.exports = router;
```

2 Exercício

2.1 API Universidade

Começamos por criar uma nova API de forma semelhante à utilizada para a `TodoItemsAPI` (Secção 1.1.2¹⁹), executando, no terminal do VSCode:

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

No ficheiro `Properties/launchSettings.json`, mudar o conteúdo da linha:

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

para

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

De seguida, adicionar os restantes packages necessários:

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

O objetivo desta API é guardar informação sobre uma Universidade, composta por entidades de diversos tipos:

- **Curso:** caracterizado por um `Id` (`long`), uma sigla (`string`) e um nome (`string`)
- **Aluno:** caracterizado por um `Id` (`long`), um nome (`string`), um curso (`Curso`) e a lista de unidades curriculares (`ICollection<UnidadeCurricular>`) a que está inscrito
- **UnidadeCurricular:** caracterizada por um `Id` (`long`), uma sigla (`string`), um nome (`string`), um curso (`Curso`) e um ano (`int`)
- **Nota:** caracterizada por um `Id` (`long`), um valor (`int`), a unidade curricular a que diz respeito (`UnidadeCurricular`), bem como o aluno ao qual está associada (`Aluno`).

É necessário criar a pasta `Models` para incluir os modelos (classes) associados a cada uma das entidades, os DTOs respetivos (quando necessário), e também o ficheiro `UniversidadeContext.cs`. À medida que formos implementando modelos, para gerar o *controller* respetivo (responsável pela implementação das rotas, referidas mais abaixo), iremos usar o comando:

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

No comando, `XXX` e `YYY` irão ser substituídos pelos nomes do modelo e do *controller*, respetivamente. Como referido, a API vai implementar diversas rotas, associadas a cada um dos modelos/*controllers*. Este projeto será executado em diversas fases, associadas aos modelos e explicadas nas secções seguintes.

? Começar por esquematizar a nova API

⚠ A API terá apenas um context, que lida com todas as entidades

¹⁹ Analisar cuidadosamente as diferenças entre estes comandos e os executados para a primeira API

2.1.1 Curso

Models/Curso.cs Um Curso é caracterizado por:

- Id (do tipo long)
- Sigla (do tipo string), a sigla do Curso
- Nome (do tipo string), o nome do Curso

UniversidadeContext.cs Este código irá definir o contexto da API, o seu construtor, e ainda que o contexto contém uma lista de cursos. É necessário também alterar o ficheiro `Program.cs`, adaptando as alterações efetuadas no caso da `TodoItemsAPI`.

CursoController.cs De forma a gerar o *controller* para o curso, deve executar-se o comando:

```
dotnet-aspnet-codegenerator controller
-name CursoController -async -api
-m Curso -dc UniversidadeContext -outDir Controllers
```

Para o curso, pretendemos implementar as rotas apresentadas na Tabela 3.

? Preencher a tabela com a informação em falta

Tabela 3: Rotas da API Universidade - Entidade Curso.

Rota	Anotação	Método	Descrição
GET: api/cursos			Permite obter todos os cursos existentes no repositório
GET: api/cursos/1			Permite obter um curso por id relacional
GET: api/cursos/LEI			Permite obter um curso por sigla
POST: api/cursos			Permite a criação de um novo curso
PUT: api/cursos/1			Permite a edição de um curso
DELETE: api/cursos/1			Permite a eliminação de um curso

As rotas descritas na tabela são semelhantes às implementadas na API anterior. A maior diferença neste caso é que temos a rota GET: `api/cursos/LEI`, que:

- Tem uma rota "parecida" com a GET: `api/cursos/1`²⁰
- Vai obter o curso a partir da sigla, que não é o id relacional²¹

²⁰Para mais informação sobre *routing* em .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>

Caso se altere o nome do método que faz GET por Id, é preciso também alterar na última linha do método que controla o POST.

²¹Ver os links da nota de rodapé anterior.

A Tabela 4 contém exemplos de pedidos a fazer à API, pela ordem em que devem ser efetuados.












 Preencher a tabela com a informação em falta

Tabela 4: Corpo dos pedidos a enviar à API e das respostas obtidas - Entidade Curso.

Rota	Corpo do Pedido	Corpo da Resposta
POST: api/cursos	{ "sigla": "LEI", "nome": "L E I" }	
POST: api/cursos	{ "sigla": "LEGI", "nome": "Lic. Eng. Gest. Ind." }	
POST: api/cursos		{ "id": 3, "sigla": "LSIG", "nome": "Lic. Sist. Inf. Gest." }
PUT: api/cursos/1	{ "id": 1, "sigla": "LEI", "nome": "Lic. Eng. Informática" }	
GET: api/cursos		
GET: api/cursos/1		
GET: api/cursos/LES		

2.1.2 Aluno

Aluno.cs Um Aluno é caracterizado por:

- Id (do tipo long)
- Nome (do tipo string), o nome do Aluno
- Curso (do tipo Curso), o Curso a que o Aluno está inscrito
- Saldo (do tipo double, o saldo do Aluno (saldo inicial: €100)
- Email (do tipo string), o email do Aluno

Atualizar UniversidadeContext.cs O contexto da API terá de passar também a incluir uma lista dos alunos.

AlunosController.cs Criar o *controller* baseado no modelo Aluno.

AlunoDTO.cs Não queremos que, ao inserir um aluno, seja obrigatório introduzir toda a informação sobre o curso (Id, Sigla, Nome); ou que, ao visualizar um aluno, apareça toda a informação do seu curso. Seria mais fácil para o utilizador que lhe fosse permitido introduzir ou visualizar apenas, por exemplo, a sigla do curso respetivo. Assim, teremos de criar um DTO para aluno, sendo que um AlunoDTO é caracterizado por:

- Id (do tipo long)
- Nome (do tipo string), o nome do Aluno
- SiglaCurso (do tipo string), a sigla do Curso a que o Aluno está inscrito
- Saldo (do tipo double, o saldo do Aluno (saldo inicial: €100)
- Email (do tipo string), o email do Aluno

Será ainda necessário atualizar o *controller* para utilizar o AlunoDTO em vez do Aluno²². No final, será possível utilizar as rotas apresentadas na Tabela 5.


 Preencher a tabela com a informação em falta

Tabela 5: Rotas da API Universidade - Entidade Aluno.

Rota	Anotação	Método	Descrição
GET: api/alunos			Permite obter todos os alunos existentes no repositório
GET: api/alunos/1			Permite obter um aluno por id relacional
POST: api/alunos			Permite a criação de um novo aluno
PUT: api/alunos/1			Permite a edição de um aluno
DELETE: api/alunos/1			Permite a eliminação de um aluno

²²Consultar documentação em <https://learn.microsoft.com/en-us/ef/ef6/ querying/related-data> e <https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-4>

A Tabela 6 contém exemplos de pedidos a fazer à API, pela ordem em que devem ser efetuados.









 Preencher a tabela com a informação em falta

Tabela 6: Corpo dos pedidos a enviar à API e das respostas obtidas - Entidade Aluno.

Rota	Corpo do Pedido	Corpo da Resposta
POST: api/alunos	{ "nome": "Berto", "siglcurso": "LEI" }	
POST: api/alunos	{ "nome": "Ana", "siglcurso": "LEGI" }	{ "id": 2, "nome": "Ana", "siglcurso": "LEGI" }
POST: api/alunos		{ "id": 3, "nome": "Carlos", "siglcurso": "LSIG" }
PUT: api/alunos/1	{ "id": 1, "nome": "Alberto", "siglcurso": "LEI" }	
GET: api/alunos		
GET: api/alunos/1		

2.1.3 UnidadeCurricular

UnidadeCurricular.cs Uma UnidadeCurricular é caracterizada por:

- Id (do tipo long)
- Sigla (do tipo string), a sigla da UnidadeCurricular
- Nome (do tipo string), o nome da UnidadeCurricular
- Curso (do tipo Curso), o Curso a que pertence a UnidadeCurricular
- Ano (do tipo int), o ano do Curso em que se insere a UnidadeCurricular

Atualizar UniversidadeContext.cs O contexto da API terá de passar também a incluir uma lista dos unidades curriculares.

UCsController.cs Criar o *controller* baseado no modelo UnidadeCurricular.

UC_DTO.cs Pelo mesmo motivo que para o Aluno, necessitamos de criar um DTO para UnidadeCurricular, que é caracterizado por:

- Id (do tipo long)
- Sigla (do tipo string), a sigla da UnidadeCurricular
- Nome (do tipo string), o nome da UnidadeCurricular
- SiglaCurso (do tipo string), a sigla do Curso a que pertence a UnidadeCurricular
- Ano (do tipo int), o ano do Curso em que se insere a UnidadeCurricular

Será ainda necessário atualizar o *controller* para utilizar o UC_DTO em vez de UnidadeCurricular. No final, será possível utilizar as rotas apresentadas na Tabela 7.

? Preencher a tabela com a informação em falta

Tabela 7: Rotas da API Universidade - Entidade UnidadeCurricular.

Rota	Anotação	Método	Descrição
GET: api/ucs			Permite obter todas as unidades curriculares existentes no repositório
GET: api/ucs/1			Permite obter a unidade curricular por id relacional
GET: api/ucs/SINF2			Permite obter a unidade curricular pela sigla
POST: api/ucs			Permite a criação de uma nova uc
PUT: api/ucs/1			Permite a edição de uma uc
DELETE: api/ucs/1			Permite a eliminação de uma uc

A Tabela 8 contém exemplos de pedidos a fazer à API, pela ordem em que devem ser efetuados.







 Preencher a tabela com a informação em falta

Tabela 8: Corpo dos pedidos a enviar à API e das respostas obtidas - Entidade UnidadeCurricular.

Rota	Corpo do Pedido	Corpo da Resposta
POST: api/ucs	<pre>{ "sigla": "LTW", "Nome": "Tecnologias Web", "siglcurso": "LEI", "ano": 2 }</pre>	
POST: api/ucs	<pre>{ "sigla": "ALPROGI", "Nome": "Algo. e Prog LEI", "siglcurso": "LEI", "ano": 1 }</pre>	<pre>{ "id": 2, "sigla": "ALPROGI", "Nome": "Algo. e Prog. LEI", "siglcurso": "LEI", "ano": 1 }</pre>
POST: api/ucs	<pre>{ "sigla": "ALPROGII", "Nome": "Algo. e Prog. LEGI", "siglcurso": "LEGI", "ano": 1 }</pre>	<pre>{ "id": 3, "sigla": "ALPROGII", "Nome": "Algo. e Prog. LEGI", "siglcurso": "LEGI", "ano": 1 }</pre>
POST: api/ucs		<pre>{ "id": 4, "sigla": "POO", "Nome": "Prog. Or. Obj.", "siglcurso": "LEI", "ano": 1 }</pre>
PUT: api/ucs/1	<pre>{ "id": 1, "sigla": "LTW", "Nome": "Lab. de Tec. Web", "siglcurso": "LEI", "ano": 2 }</pre>	
GET: api/ucs		

Continua na próxima página

Tabela 8 – Corpo dos pedidos a enviar à API e das respostas obtidas - Entidade UnidadeCurricular (cont.)

Rota	Corpo do Pedido	Corpo da Resposta
GET: api/ucs/1	?	?
GET: api/ucs/LTW	?	?

2.1.4 Nota

Nota.cs Uma Nota é caracterizada por:

- Id (do tipo long)
- Valor (do tipo double), o valor da Nota obtida, de 0.0 a 20.0
- UnidadeCurricular (do tipo UnidadeCurricular), a UnidadeCurricular a que diz respeito a Nota
- Aluno (do tipo Aluno), o Aluno a que pertence a Nota nessa UnidadeCurricular

Atualizar UniversidadeContext.cs O contexto da API terá de passar também a incluir uma lista das notas.

NotasController.cs Criar o *controller* baseado no modelo Nota.

Nota.DTO.cs Pelo mesmo motivo que para o Aluno e UnidadeCurricular, necessitamos de criar um DTO para Nota, que é caracterizado por:

- Id (do tipo long)
- Valor (do tipo double), o valor da Nota obtida, de 0.0 a 20.0
- SiglaUC (do tipo string), a sigla da UnidadeCurricular a que diz respeito a Nota
- NomeAluno (do tipo string), o nome do Aluno a que pertence a Nota nessa UnidadeCurricular

Será ainda necessário atualizar o *controller* para utilizar o Nota.DTO em vez de Nota. No final, será possível utilizar as rotas apresentadas na Tabela 9.

Preencher a tabela com a informação em falta

Tabela 9: Rotas da API Universidade - Entidade Nota.

Rota	Anotação	Método	Descrição
GET: api/notas			Permite obter todas as notas existentes no repositório
GET: api/notas/1			Permite obter as notas de um aluno pelo id do aluno
GET: api/notas/ALPROGII			Permite obter as notas de uma unidade curricular pela sigla da unidade curricular
POST: api/notas			Permite a criação de uma nova uc
PUT: api/notas/1			Permite a edição de uma uc
DELETE: api/notas/1			Permite a eliminação de uma uc

A Tabela 10 contém exemplos de pedidos a fazer à API, pela ordem em que devem ser efetuados.

Preencher a tabela com a informação em falta

Tabela 10: Corpo dos pedidos a enviar à API e das respostas obtidas - Entidade Nota.

Rota	Corpo do Pedido	Corpo da Resposta
POST: api/notas	{ "valor":10.0, "nomealuno":"Alberto", "siglauc":"ALPROGI" }	?
POST: api/notas	{ "valor":15.0, "nomealuno":"Carlos", "siglauc":"ALPROGII" }	{ "id":2, "valor":15.0, "nomealuno":"Carlos", "siglauc":"ALPROGII" }
POST: api/notas	{ "valor":11.0, "nomealuno":"Ana", "siglauc":"P00" }	{ "id":3, "valor":11.0, "nomealuno":"Ana", "siglauc":"P00" }
PUT: api/notas/1	{ "id": 1, "valor": 12, "nomeAluno": "Alberto", "siglaUC": "ALPROGI" }	?
GET: api/notas	?	?
GET: api/notas/1	?	?
GET: api/ucs/ALPROGI	?	?

2.1.5 Migrar para uma BD SQLite

De acordo com o referido anteriormente na secção [1.1.11](#).

2.1.6 Possíveis melhorias

Possíveis melhorias que poderiam ser adicionadas à API (entre outras):

- Implementar a rota GET: `api/aluno/EI`, que obtém todos os alunos inscritos na licenciatura cuja sigla é dada
- Acrescentar uma lista de unidades curriculares ao aluno
- Ao inscrever um aluno numa UC, validar que a UC é do curso em que o aluno está inscrito
- Ao inserir uma nota, validar que o aluno está inscrito nessa UC
- ...

2.2 API Cantina

De forma a guardar informação sobre uma Cantina, é necessário construir uma API NodeJS:

- Com recurso ao `express-generator`²³
- Utilizando uma base de dados MongoDB, alojada no Atlas mongoDB²⁴
- Que comunica com a API C# criada anteriormente, como referido na secção 2.2.3

Abaixo refere-se a informação que deve ser guardada, juntamente com as rotas que é necessário implementar na API.

2.2.1 Prato do dia

Um prato do dia é caracterizado por:

- `id` (do tipo `Schema.Types.ObjectId`)
- `nome_prato` (do tipo `String`), o nome do prato
- `dia_prato` (do tipo `String`), o dia a que corresponde o prato

As rotas a implementar, associadas à entidade *Prato do dia*, são apresentadas na Tabela 11.

Tabela 11: Rotas da API Cantina - Entidade Prato do dia.

Rota	Verbo HTTP	Descrição
<code>/api/prato</code>	POST	criar um PratoDoDia
<code>/api/prato</code>	GET	listar todos os PratoDoDia
<code>/api/prato/28412548h2123</code>	GET	listar um PratoDoDia
<code>/api/prato/28412548h2123</code>	PUT	editar um PratoDoDia
<code>/api/prato/28412548h2123</code>	DELETE	remover um PratoDoDia

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

²⁴<https://www.mongodb.com/>

2.2.2 Ementa da semana

Uma ementa da semana é caracterizada por:

- `id` (do tipo `Schema.Types.ObjectId`)
- `data` (do tipo `Date`), a data a que corresponde a ementa, com valor `default Date.now`
- `listaPratos` (uma lista de objetos do tipo `Prato`), a lista de pratos do dia dessa semana

As rotas a implementar, associadas à entidade *Ementa da semana*, são apresentadas na Tabela 12.

Tabela 12: Rotas da API Cantina - Entidade Ementa da semana.

Rota	Verbo HTTP	Descrição
/api/ementa	POST	criar uma ementa
/api/ementa	GET	listar todos as ementas
/api/ementa/28412548h2123	GET	listar uma ementa
/api/ementa/28412548h2123	DELETE	remover uma ementa

2.2.3 Reserva

Uma reserva é caracterizada por:

- `id` (do tipo `Schema.Types.ObjectId`)
- `data` (do tipo `Date`), a data a que corresponde a reserva, com valor `default Date.now`
- `pratoReservado` (do tipo `Prato`), o prato reservado
- `aluno` (um objeto que guarda `Num_aluno`, `Nome_aluno` e `Email_aluno`)

As rotas a implementar, associadas à entidade *Reserva*, são apresentadas na Tabela 13.

Tabela 13: Rotas da API Cantina - Entidade Reserva.

Rota	Verbo HTTP	Descrição
/api/reserva	POST	criar uma reserva
/api/reserva	GET	listar as reservas
/api/reserva/1980980	GET	listar as reservas de um aluno
/api/reserva/1980980	DELETE	remover as reservas de um aluno

Ao **criar uma reserva**, é necessário ligar à API feita em C#, usando `node-rest-client`²⁵, para obter os dados do aluno e atualizar o seu saldo (assumir que o preço da reserva é de €4.00). Terá de ser criada uma nova rota capaz de atualizar o saldo aluno aquando da reserva de uma refeição.

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