REST Web Services

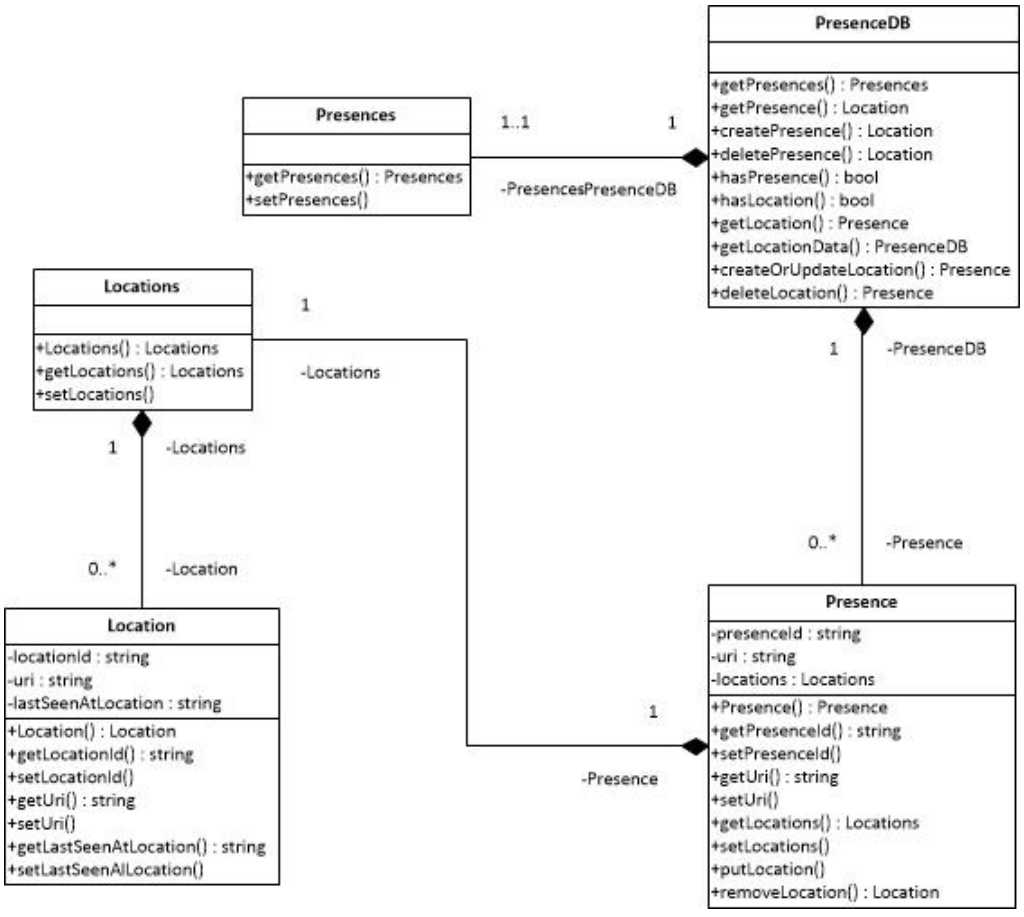
2019/2020

Description

This exercise provides an introduction to Web services programming in C# using .NET core Web API framework and Visual Studio Code. In this exercise we are going to develop a presences REST application. A presence is identified by a presence ID (a string – IP – that identifies an entity that can be a user or a machine). A presence may also have associated a list of locations. A location is identified by a location ID and has also associated a timestamp that corresponds to the time the entity was last seen at that location. In this prototype we do not support either entity (user or machine) creation or entity authentication. We assume that only creators of presences may update presences' information and location. A proper authentication's functionality (based on user or machine authentication) should then be provided in a final system. Our application supports the operations of creating, updating, consulting and deleting presences and locations. We neither support concurrent access. We should analyse concurrency issues and suggest and discuss a solution with your teacher.

Application architecture/Logic

The class diagram depicted in the following picture describes the application logic. The `PresencesDB` class encapsulates the Presences service operations. These are the C# operations that expose the application resources. The code is provided in the `presencesRESTService.zip` file (*Models* folder).



Resources representations and service operations

In order to design a REST application, we must expose the service data mode as resources and define the service operations (based on HTTP methods).

HTTP Method	Resource URI	Request body	Response

POST	/presences
	
	Creates a new presence resource. It uses presenceId in the resource representation to create the new resource URL; if it already exists, it does not create a new resource
		Contains the representation of presence resource in JSON.
		Parameters:
		`presenceId` – the identification of the presence.
		
		Example request
		
		`POST …/presences HTTP 1.1`
		`Content-type: application/json`
		`{`
		`  "presenceId" : "193.137.1.4"`
		`}`
			201 Created, the request has been fulfilled and resulted in a new presence resource being created, response contains an entity describing the resourced created
			
			204 No Content, the server has successfully processed the request, but is not returning any content; the resource has not been created because it already existed
			
			400 Bad Request, the request cannot be fulfilled due to bad syntax
			
			……

GET	/presences Retrieves the list of presences	Request body is empty Example request `GET …/presences HTTP 1.1`	200 OK, response contains an entity corresponding to the resource requested ----
GET	/presences/{presenceId} Retrieves the presence resource	Request body is empty Example request `GET …/presences/193.137.1.1 HTTP 1.1`	200 OK, response contains an entity corresponding to the resource requested 404 Not Found, the requested resource could not be found

DELETE	/presences/{presenceId} Deletes the presence resource	Request body is empty Example request `DELETE …/presences/193.137.1.1 HTTP 1.1`	204 No Content, the server has successfully processed the request, but is not returning any content; 404 Not Found, the request resource could not be found
GET	/presences/{presenceId}/locations Retrieves the locations resource associated to presence resource	Request body is empty Example request `GET …/presences/193.137.1.1/locations HTTP 1.1`	200 OK, response contains an entity corresponding to the resource requested 404 Not Found, the requested resource could not be found

PUT	/presences/{presenceId}/locations/{locationId}
	
	Updates a location resource. If the resource does not exist, the server creates a new resource; It uses presenceId in the resource representation to create the new resource URL; If the resource exists, the resource is updated with new representation
		Contains the representation of location resource in JSON.
		Parameters:
		
		`locationId` – the identification of the location.
		
		`lastSeenAtLocation` – the time the entity was last seen at this location in text format "`MMM dd,yyyy HH:mm`"
		
		Example request
		`POST …/presences/193.137.1.1/locations/bar HTTP 1.1`
		`Content-type: application/json`
		
		`{`
		`"lastSeenAtLocation" : "Apr 27,2015 16:33",`
		`" locationId" : "Azurem"`
		`}`
			200 Ok, the request has been fulfilled and resulted in an updated resource, response contains an entity describing the new resource representation
			
			404 Not Found, the container resource could not be found
			
			400 Bad Request, the request cannot be fulfilled due to bad syntax

GET	/presences/{presenceId}/locations/{locationId} Retrieves the location resource associated to presence resource	Request body is empty Example request `GET …/presences/193.137.1.1/locations/bar HTTP 1.1`	200 OK, response contains an entity corresponding to the resource requested 404 Not Found, the requested resource could not be found
DELETE	/presences/{presenceId}/locations/{locationId} Deletes de location resource	Request body is empty Example request `DELETE …/presences/193.137.1.1/locations/bar HTTP 1.1`	204 No Content, the server has successfully processed the request, but is not returning any content; 404 Not Found, the request resource could not be found

Resource representation examples

Presence resource


```
{
   "presenceId": "193.137.1.2",
   "locations": {
        "locations": [
              {
                 "lastSeenAtLocation": "Apr 27,2018 16:33",
                 "locationId": "DSI"
              },
              {
                 "lastSeenAtLocation": "Apr 27,2018 16:50",
                 "locationId": "LAP5"
              }
         ]
     }
}
```


Presences resource


```
{
    "presences": [
        {
            "presenceId": "193.137.1.1",
            "locations": {
                "locations": [
                    {
                        "lastSeenAtLocation": "Apr 27,2018 16:33",
                        "locationId": "DSI"
                    },
                    {
                        "lastSeenAtLocation": "Apr 27,2018 16:50",
                        "locationId": "LAP5"
                    }
                ]
            }
        },
     ……..
    ]
}
```

Locations resource	{
	"locations": [
	{
	"lastSeenAtLocation": "Apr 27,2018 16:33",
	"locationId": "DSI"
	},
	{
	"lastSeenAtLocation": "Apr 27,2018 16:50",
	"locationId": "LAP5"
	}
	]
	}

Location resource	{
	"lastSeenAtLocation" : "Apr 27,2018 16:33",
	" locationId" : "Azurem"
	}

Web service development and deployment using .NET core, Visual Studio Code and Docker

Tools requirements

Before you start, you will need to install the following tools:

- .NET core (https://dotnet.microsoft.com/download/dotnet-core)
- Visual Studio Code (https://code.visualstudio.com/)
- C# for Visual Studio Code (latest version)
- Docker Desktop (https://www.docker.com/products/docker-desktop)
- Postman tool

Tasks

Create a new project

T1 Open the integrated terminal (View -> Terminal). Type "*mkdir presencesRESTService*", and again type "*dotnet new webapi*", to create a .NET Core Web API template. Finally type "*dotnet restore*[1]".

Some files are generated by template:

`Program.cs`

The *program.cs* contains the main static method (the starting point of the program). The main method of a Web API application template creates a new webhost builder and then runs the webhost to start listening to the requests. This file It also instantiates a class called *startup*.

`Startup.cs`

The *startup* class is where to setup how to answer the requests.

Code the Web service logic

Before you implement the Web Service, you should first implement the application logic.

T2 Add the model classes to your project. You may add the model classes to a folder *Models*[2] in the project folder. Add the five classes in file `presencesRESTService.zip` (*Models* folder) to the *Models* folder of your project.

Create the web APIs

To handle requests, a REST web service uses controllers[3]. Each browser request is mapped to a particular controller. The controller is responsible for generating the response to the browser request. In the context of Web APIs, the controller usually will return a HTTP response to the caller (in some cases, the controller may redirect the call to another controller).

A controller exposes controller actions. An action is a method on a controller that gets called when you enter a particular URL in your browser address bar. A controller action must be a public method of a controller class.

A controller action returns something called an action result. An action result is what a controller action returns in response to a browser request. .NET Core offers several options for web API controller action return types[4]: `Specific type`, `IActionResult` and `ActionResult<T>`.

---

[1] To promote a cleaner development environment and to reduce repository size, NuGet Package **Restore** installs all of a project's dependencies listed in either the project file or packages.config. The .NET Core 2.0+ **dotnet** build and **dotnet** run commands **do** an automatic package **restore** (https://docs.microsoft.com/en-us/nuget/consume-packages/package-restore).
[2] Model classes can go anywhere in the project, but the *Models* folder is used by convention.
[3] https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs
[4] https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-3.0

.NET Core also has support for formatting response data. Response data can be formatted using specific formats or in response to client requested format.

The process of mapping incoming HTTP requests to particular controller actions is the routing process When you create a new Web API .NET application, the application is already configured to use .NET Routing[5]. There are two types of routing for action methods: Conventional Routing and Attribute Routing.

Using the conventional type, the default route template is configured as `{controller=Home}/{action=Index}/{id?}`. This will match the `Index()` method in `HomeController` with an optional parameter `id` by default. This can also match a URL path like `/Books/Details/5` and will extract the route values `{controller = Books, action = Details, id = 5}` by tokenizing the path. .Net core will attempt to locate a controller named `BooksController` and run the action method `Details()` by passing the `id` parameter as 5[6].

Using the attribute type, you may use the `[Route]` and `[Http[Verb]]` attributes to associate URL paths to controllers and actions.

T3 Create a controller and actions to answer the HTTP requests. The URLs and formats of the application resources have been defined in section (**Resources representations and service operations**). The Web API application template provides a demo controller: the WeatherForecastController.cs. You may use the .NET tools to generate a controller template with template actions for your application:

- Go to the project folder;
- Install the dotnet-aspnet-codegenerator tool. For this, you should run the following commands:
  - dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
  - dotnet add package Microsoft.EntityFrameworkCore.Design
  - dotnet tool install --global dotnet-aspnet-codegenerator
- apply the command:

```
dotnet aspnet-codegenerator controller -name presencesController -api -actions -outDir Controllers
```

This command generates a controller template for your Web API.

Implement the Web service

T4 Copy the content of the `presencesController.cs` file provided in the `presencesRESTService.zip` (*Controllers* folder) file to the new generated `presencesController.cs`. This controller exposes the controller actions that get called to answer the HTTP requests (defined above in section **Resources representations and service operations**). The following table maps the HTTP interface with the presences resource (`/presences`) and its sub-resources (`/presences/{presenceId}`, `/presences/{presenceId}/locations` and `/presences//{presenceId}//locations/{locationId}`).

[5] https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/asp-net-mvc-routing-overview-cs

[6] https://code-maze.com/routing-asp-net-core-mvc/

HTTP Method	Resource URI	Controller action
POST	/presences	`public ActionResult<Presence> PostPresence([FromBody] Presence p)`
GET	/presences	`public ActionResult<Presences> GetPresences()`
GET	/presences/{presenceId}	`public ActionResult<Presence> GetPresence(String presenceId)`
DELETE	/presences/{presenceId}	`public ActionResult DeletePresence(String presenceId)`
GET	/presences/{presenceId}/locations	`public ActionResult<Locations> GetLocations(String presenceId)`
PUT	/presences/{presenceId}/locations/{ locationId }	`public ActionResult<Location> PutLocation([FromBody] Location loc, String presenceId, String locationId)`
GET	/presences/{presenceId}/locations/{locationId}	`public ActionResult<Location> GetLocation(String presenceId, String locationId)`
DELETE	/presences/{presenceId}/locations/{locationId}	`public ActionResult DeleteLocation(String presenceId, String locationId)`

The attribute `[ApiController]` defines the controller as Web API: actions will return a media format.

The attribute `[Route("api/[controller]")]` defines the base route for this controller as "`api/presences`" (presences is the name of the controller).

The attribute `[HttpGet]` indicates that the action will get called for a GET method.

The attribute `[Route]` defines which URL should be indicated. For example, the attribute `[Route("{presenceId}")]`, defines that `GetPresence()` is the action called when a GET on "`api/presences/{presenceId}`" is received.

Parameter binding is another important process in .NET core We API applications. The action `GetPresence()` extracts the `String presenceId` input parameter form the URL. The action `PostPresence()` extracts the Controller actions input parameters may either be extracted from the URL the `Presence p` input parameter form the request body.

The return type of the presences controller actions is `ActionResult<T>` type. .NET Core automatically serializes the object to json and writes the json into the body of the response message. The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetPresence` can return two different status values:

- If no item matches the requested ID, the method returns a `404 NotFound` error code.
- Otherwise, the method returns 200 OK.

The `CreatedAtAction` method is used to return the result of action `PostPresences`:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The Location header specifies the URI of the newly created to-do item.
- Indicates the `GetPresences()` action to create the Location header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Test the Web service

T5 In the integrated terminal, in the project folder, type "*dotnet run*" to run the REST web service.

Your service base URL should be (if running on *localhost*): http://localhost:5001/api/presences.

Use "Postman" application and test your Web service. Please see request examples above.

Deploy the Web service with Docker

(PART 2)