POLITÉCNICO DE LEIRIA
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

www.dei.estg.ipleiria.pt

Systems Integration

RESTful Web Services and the ASP.NET Web API

Scope:
- Background
- URIs following directory structure and message contents
- Creating a RESTful Web Service (ASP.NET Web API)
- Example and exercise
- Create a database in SQL Server
- CRUD operations in a SQL Server Database
- Consume a RESTful API (web and desktop client)
- Document and Test a RESTful Web API service (using Swagger)

Duration: 2 class

Background

SOAP-based Web Services (although very powerful) present some drawbacks to the users/programmers, such as, the overhead added to the messages, the need for a client proxy to invoke the service, powerful XML parsers, the different ways a service could be modeled in a user API, etc.

One of the main aims of RESTful Web Services is to standardize the services API design by relying solely in Web Technologies, such as URIs, HTTP and XML/JSON (and stateless). For instance, RESTful guide developers to employ HTTP main verbs (GET, POST, PUT, DELETE) to achieve the same as when using Database CRUD operations (Create, Retrieve, Update, Delete) with the following correspondences:

Database Verb	**HTTP Verb**
Create	Post
Retrieve	Get
Update	Put
Delete	Delete

But is it mandatory to adopt HTTP verbs? Can we use solely GET verb and issue data retrieval, deletion, creation, and updates operations? Yes, you can use solely GET verb and use query strings to pass arguments, but this is not what RESTful defines. Proceeding by this way, we will reach the SOAP-based non-standard way to define services API…

Evidence of RESTful web services popularity is the adoption of RESTful technology by World Wide Web 2.0 service providers including Facebook, Google, Yahoo, etc.

On the other hand, RESTful Web Services are stateless and 100% Web compatible. This means that a RESTful Web Services could be invoked directly from Browser (unlike SOAP-base Web services) and service scalability is completely transparent in the presence of a server cluster infrastructure.

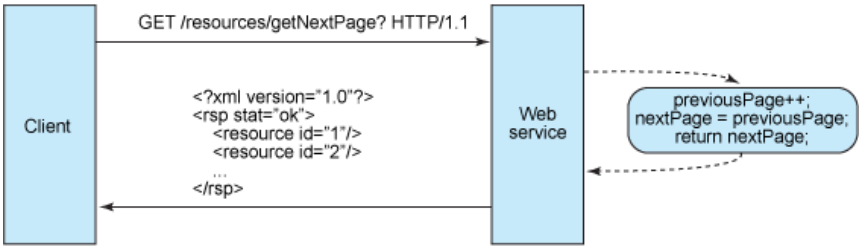


Figure 1. Stateful design [IBM]

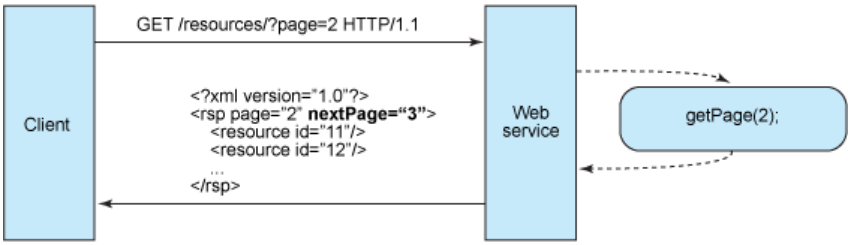


Figure 2. Stateless design [IBM]

Here are some examples of HTTP verbs and URLs (calls) in the context of RESTful Web Services:

GET	`http://<domain>/customers`	Gets all resources
GET	`http://<domain>/customers/6`	Get resource for ID 6
POST	`http://<domain>/customers`	Creates new resource. Details in the HTTP body
PUT	`http://<domain>/customers/5`	Updates resource with ID 5. Details in the HTTP body
DELETE	`http://<domain>/customers/5`	Deletes resource with ID 5

REST was first described by Roy Fielding in his doctoral dissertation, *Architectural Styles and the Design of Network-based Software Architectures*:

`http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

Visual Studio offers two main ways to create RESTful Web Services: WCF and ASP.NET Web API. WCF is the Visual Studio framework to create services that support multiple transport protocols (HTTP, TCP, UDP, and custom transports) and allows switching between them, where SOAP-based Web Services are included, while ASP.NET Web API is focused solely on HTTP services in a high-level approach. This worksheet is devoted to ASP.NET Web API and how to create RESTful Web Services on it.

1. URIs following directory structure and message contents

In order, too easy the client-side call, URIs on the server side should be structured like filesystems directory structures. Take the example of a discussion forum service [IBM]:

`http://www.myservice.org/discussion/topics/{topic}`

In this example, we could see a root node called discussion and a topics node beneath it. Inside topics node there can be any type of discussions threads such as database, hardware, etc. Within this structure, it's easy to pull up discussion threads just by typing something after /topics/ [IBM]. Another option to define the discussion topics could be from a date perspective [IBM]:

`http://www.myservice.org/discussion/2019/10/01/{topic}`

So, it is extremely important to identify all resources behind the service and structure them in a hierarchical directory approach.

Once set up in a directory structure, the RESTful service could be invoked. For both service invocation and response, XML or JSON format could be used. XML is more suitable for high structure data while JSON (JavaScript Object Notation) is used when XML (complex) parsing must be avoided (e.g. lack of resources such as in mobile and wearable devices).

The **XML** approach:

```
<product>
     <id>1</id>
     <name>Laptop Asus</name>
     <description>ASUS ZENBOOK UX305FA. Fast. Slim. Beautiful. </description>
</product>
```

The **JSON** approach:

```
"product" :
{
     "id" : 1,
     "name" : "Laptop Asus ",
     "description" : " ASUS ZENBOOK UX305FA. Fast. Slim. Beautiful.",
}
```

2. Creating a RESTful Web Service (ASP.NET Web API)

Create a new project (and solution) of type ASP.NET Web Application (.NET Framework), with the name **ProductsAPI** [www.asp.net] and then choose **Empty Template** and select solely **Web API** (see Figure 1).

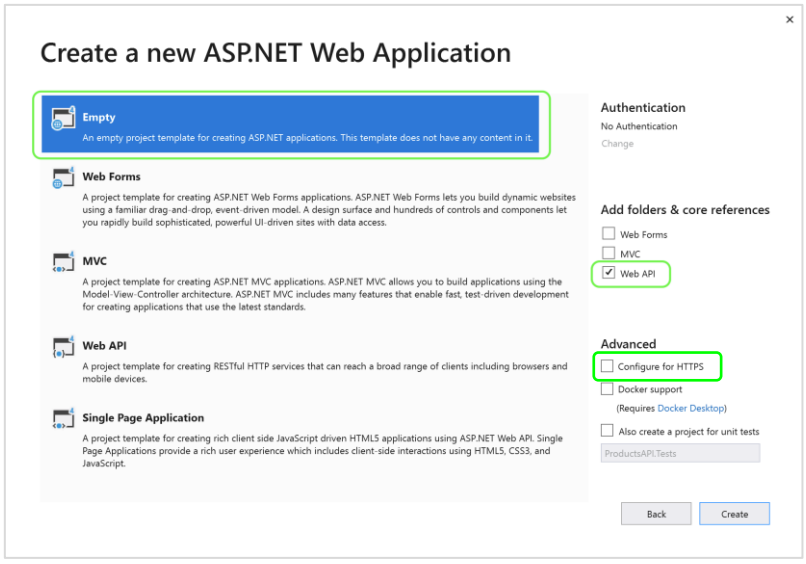


Figure 1 – Project template settings when creating the ASP.NET Web Application.

Add a model (represents data and can be serialized) for product by right-clicking model and add new class Product.

```
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

Add now a Controller (handles http requests) using Add | Controller, of type **Web API 2 Controller – Empty**, using name **ProductsController** (do not touch word Controller).

> **Alternative/Tip:** When adding a new controller, you may choose to create it with all the CRUD actions, therefore you may choose the **Web API 2 Controller with read/write actions** instead of the *Web API 2 Controller – Empty template*.

You don't need to put your controllers into a folder named Controllers. The folder name is just a convenient way to organize your source files.

Replace the **ProductsController** class source code with the following one:

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsAPI.Controllers
{
    public class ProductsController : ApiController
    {
        //Probably a database in a real scenario...
        List<Product> products = new List<Product>
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product); //Respecting HTTP errors (200 OK)
        }
    }
}
```

The used method names map automatically to one or more URIs (naming convention):

GetAllProducts	`/api/products`	**Using http GET**
GetProduct	`/api/products/id`	**Using http GET**

For more information about how Web API routes HTTP requests to controller methods, see http://www.asp.net/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api.

Now, run the Web API project. Probably you will see a **HTTP Error 403.14 – Forbidden** error. This means that the Web API project doesn't have an associated default viewable page (ASPX, HTML, etc.). Add `/api/products` or `/api/products/1` to the address bar to access web API.

[Extra/Optional] Create a simple JavaScript client:

Now, let's call the created Web API using JQuery in JavaScript:

Right click project, add, new item (in C#|Web), HTML page, name=**index.html**

Replace web page content by [www.asp.net]:

```
<!DOCTYPE html>
<html>
<head>
    <title>RESTful web service - Products App</title>
        <meta charset="utf-8" />
</head>
<body>
    <div>
        <h2>All Products</h2>
        <ul id="products"/>
    </div>
    <div>
        <h2>Search by Id</h2>
        <input type="text" id="prodId" size="5"/>
        <input type="button" value="Search" onclick="find();"/>
        <p id="product" />
    </div>

    <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.4.min.js"></script>
    <script>
        var uri = 'http://localhost:49760/api/products'; //TODO: MUST BE UPDATED

        $(document).ready(function ()
        {
            $.get(uri)
            .done(function (data) {
                alert(data);
                //ON SUCESS, 'data' contains a list of products
                $.each(data, function (key, item) {
                    //ADD a list item for the product
                    $('<li>', { text: formatItem(item) }).appendTo($('#products'));
                });
            })

            .fail(function (jqxhr, textStatus, error) {
                var err = textStatus + ",  " + error;
                alert("Request failed: " + err);
            });
        });

        function formatItem(item)
        {
            return item.Name + ": " + item.Price + "€";
        }

        function find()
        {
            var id = $('#prodId').val();
            $.getJSON(uri + '/' + id)
            .done(function (data) {
                $('#product').text(formatItem(data));
            })

            .fail(function (jqxhr, textStatus, error) {
                var err = textStatus2 + ", " + error;
                alert("request failed! : ", err);
            });
        } //find()

    </script>
</body>
</html>
```

Now, run again the web API application. Now, the **index.html** is shown and you can fetch some products using the form.

3. Additional Exercises

1. Besides the previous **GET** route "/api/products/1", implement a new **action** to the controller that will allow to request all the products from specific category.

   1.1. The string "api/products/{category} " is the new URI template for the route. Notice that the "{category}" parameter in the route template matches the name of the *category* value. Web API tries to match the request URI to the template. In this example, "products" is a literal segment, and "{category}" is a variable parameter. The following URIs would match this template http://<domain>/api/products/Toys[1].

```
[Route("api/products/{category}")]
public IEnumerable<Product> GetProductByCategory(string category)
```

The final GET actions (routes) will become like:

```
[Route("api/products/{id:int}")]   //specifies that the id parameter is an integer
public IHttpActionResult GetProduct(int id)

[Route("api/products/{category}")]
public IEnumerable<Product> GetProductByCategory(string category)
```

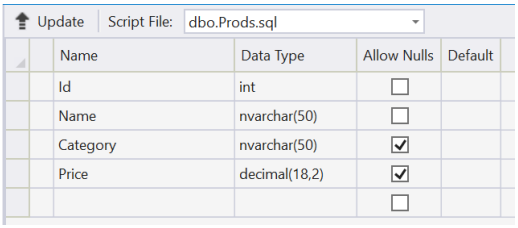
   1.2. Test the operation using the browser (only possible when testing GET) or using an API Testing tool[2].

   1.3. **[Extra]** Update also the index.html form do call the newly implemented operations.

2. Create a new version of this RESTful API. This new approach will work with a database (SQL Server Database) to store the data required by the service.

   2.1. Create a new WebAPI project/solution named **ProductsDatabaseAPI** (using the same project template as in the previous example).

   2.2. Add to the "App_Data" folder a new SQLServer database called **DBProds**. Open the empty database in the Server Explorer (double click in the DBProds.mdf file) and create a table named **Prods** with the required fields (Id, Name, Category and Price)**.**




```
CREATE TABLE [dbo].[Prods] (
    [Id]       INT        IDENTITY (1, 1) NOT NULL,
    [Name]     NVARCHAR(50) NOT NULL,
    [Category] NVARCHAR(50) NULL,
    [Price]    DECIMAL(18, 2) NULL,
);
```

   2.3. Try to clearly identify the **ConnectionString** required to access the database that you have created. Add an *ApplicationSettings* to store this information (right-click in the Project properties and Add a new Setting {name=ConnStr; type=ConnectionSTring, ...}). Use the setting in the ProductsController.cs file, e.g.:

[1] More information about routing: https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2

[2] Postman, Insomnia, etc.

```
string connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["ProductsDatabaseAPI.Properties.Settings.ConnStr"].ConnectionString;

or

string connectionString = ProductsAPIDatabase.Properties.Settings.Default.ConnStr;
```

- In the **Web.config** file change the path to database file to use the "|DataDirectory|" instead of the full path in the *AttachDbFilename* property:

```
<applicationSettings>
  <ProductsAPIDatabase.Properties.Settings>
    <setting name="ConnStr" serializeAs="String">
      <value>Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename="|DataDirectory|\DBProds.mdf";Integrated Security=True</value>
    </setting>
  </ProductsAPIDatabase.Properties.Settings>
</applicationSettings>
```

2.4. Implement the GET all products and the GET product by Id actions, that will allow us to obtain the products data from the database. For now, use the ADO.NET native approach (**SQLClient** – see note).

```
public IEnumerable<Product> GetAllProducts()
public IHttpActionResult GetProduct(int id)
```

Note: In the MSDN explore the `SqlConnection`, `SqlCommand`, `SqlDataReader` and the `SqlDataAdapter` classes from the `System.Data.SqlClient` namespace.

- They allow communication with the database and the execution of CRUD operations.
- Find examples in: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-code-examples or in MSDN online documentation.

2.5. Implement the following actions (POST, PUT and DELETE).

```
// POST api/<controller>  //our controller is products
public IHttpActionResult PostProduct([FromBody]Product p)

// PUT api/<controller>/5
public IHttpActionResult PutProduct(int id, [FromBody]Product p)

// DELETE api/<controller>/5
public IHttpActionResult DeleteProduct(int id)
```

2.6. Test the operation using the browser. (Recommendation: use chrome and install an extension to test the POST, PUT and DELETE or you may use other API testing tools like Postman[3]).

2.7. **[Extra]** Publish the service in the cloud (e.g., AppHarbor, Azure, etc.).

In .NET framework, **besides ADO.NET** classes, you may use the **ADO.NET Entity Framework**[4] to work with a database. The ADO.NET Entity Framework enables developers to create data access applications by programming against a conceptual application model instead of programming directly against a relational storage schema. The goal is to decrease the amount of code and maintenance required for data-oriented applications.

[3] Postman, Bruno, Insomnia or cURL.

[4] See **Entity Framework** (ORM) additional information: http://www.entityframeworktutorial.net/what-is-entityframework.aspx

3. A good **API documentation** is critical to provide a positive developer experience. API documentation is written text (or reference manual) that complements an API and explains how to effectively use the API. It can be created manually or automatically generated, using API documentation software.

   Swagger is a simple but powerful representation of the RESTful API. Nowadays most developers are using Swagger in almost every modern programming language and deployment environment to document.

   3.1. Add Swagger to your ASP.NET Web API project. You need to install an open-source project called **Swashbuckle** via NuGet.

   3.2. Start a new debugging session (F5 key) and navigate to **http://<domain>]/swagger** and then you should see the help pages for your APIs.

   Note: See how to personalize the generated documentation guideline here.

4. Implement a Windows Client application to consume de previous RESTFull API (**ProductsDatabaseAPI**). You can create the project from scratch (add a new Windows Forms App project named **ClientProductsApp** to the solution) or download the already created project from the course web page (option add existing project to the solution).

   4.1. Call the methods available in the API in buttons available in the form.

Note: Use the **`HttpWebRequest/Response`** class or the **`HttpClient`**[5]. You may also choose to use an external library, e.g., the **RestSharp** (install it with manage NuGet Packages).

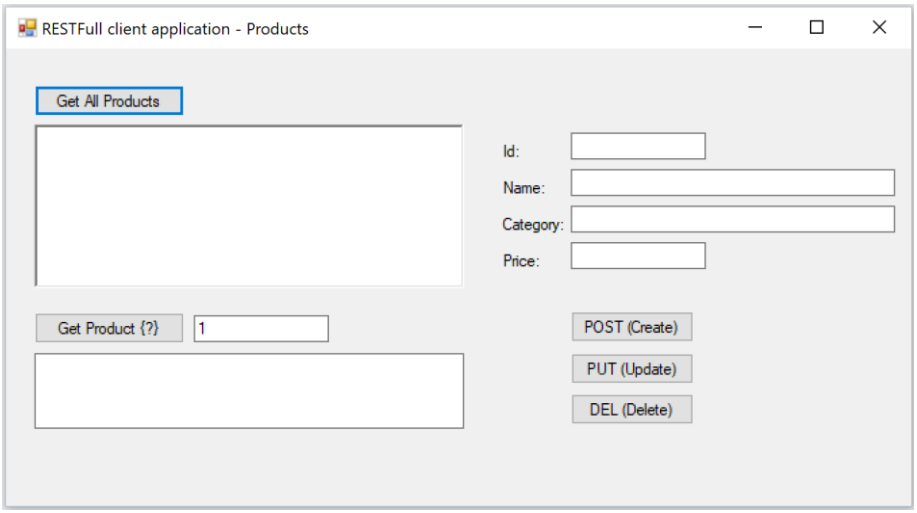


Figure 2 - Suggested interface for the ClientsProductsApp application.

Tip: See additional documentation of **RestSharp** at https://restsharp.dev to help implement the client application using the RESTful API (see how to call a GET, PUT and POST method, etc.).

[5] You may also use the **HttpClient** (https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client)