Creating Back-end Services for Mobile Applications

Ahmer Butt, Natrix

May 8, 2017

Abstract

In this article, we propose the problem of creating a mobile back-end. We will explore the solution of creating a mobile back-end as a service. This means to create a Web API to which all front-end clients can communicate with. We discuss how this API may be expanded to provide many features, such as user management and push notifications. Finally, we design and implement a simple back-end service in ASP.NET Core and discuss the limitations and depths of our solution. A GitHub repository containing our solution can be found at https://github.com/ahmerb/SampleWebAPI/.

1 Introduction

The purpose of this report is to explore and give an implementation of a back-end service for a mobile application. This is a model for providing mobile app developers a way to link their applications to back-end cloud storage and APIs exposed by back end applications [1]. This is an especially important problem for many reasons. First, a solution to it gives app developers a means to use a shared cloud-based database for their application. The cloud based server can also provide other features, such as user management, push notifications and social media integration [1]. A back-end service can provide one single API for dealing with all these services. Secondly, developers can create a single shared back-end that mobile apps, single page web applications and any other front-end client, such as a wearable device, can use. Developers do not need to recreate a new back-end each time they develop an app for a new platform. Therefore, a solution to this problem saves a hugely significant amount of time and, probably, money.

In our project, Natrix Voting, we created two mobile apps and a web app. They share a back-end. In this report, we will implement a back-end service. However, we will be focusing on how we can create a server that exposes an interface such that a mobile application can use it. We will not be creating a complex back-end that would usually be encapsulated by this. Instead, we will give the typical endpoints a small back-end service would expose and give a minimal implementation of these. For example, we will not create a cloud-based database that would be maintained by the server, but instead use an in-memory list. With these simplifications, we can focus on describing how to create a service for mobile apps to interface with, rather than the problems of implementing the back-end for an application and maintaining a cloud-based database.

In Section 2, we will be discussing the design patterns and frameworks we chose to use. In Section 3, we will present a step through guide to implementing a simple back-end service. It will provide a small subset of the features that the API of our Natrix Voting app provides. In Section 4, we will discuss the limitations of the solution presented and review the simplifications given above. We will also discuss alternative solutions. Throughout this report, to be able to focus on the problem and aid flow, we will be assuming the reader has some background in web development with the Microsoft stack.

2 Design & Approach

We will build the back-end for a simple voting application. There will be questions, which have two possible answers. Users can vote for either answer. All clients to the back-end service will use the same shared collection of questions. Clients will communicate with the server via the HTTP protocol, as standard on the web. Questions will be communicated in JSON format. The server will expose endpoints for viewing all questions, creating a question, editing a question and deleting a question. This covers the basic CRUD (Create, Read, Update, Delete) operations that a service encapsulating a database would typically expose.

We will use the MVC design pattern. The model will be our database entity, i.e. a question. We will route HTTP requests to a controller. It will choose a correct method, based on the HTTP verb given in the request, to update the list of question items appropriately and then generate and return a suitable HTTP response. The view is the mobile application. It takes the JSON data returned to it by the server and creates a consumer-facing user interface. We will not implement a mobile application in this report.

Questions will contain six fields. An id, the question text, the two choices of answer, and the current vote count for the two answers. When the server is executed, it will create a list of two example questions and store them in a list.

The controller will deal with a HTTP request, perform the required CRUD operations, and then return the correct HTTP response. To get the list of questions, the client will make a GET request. The server will return an OK (200) status code with the list of all questions. To create a question, the client will make a POST request. The server will return an OK (200) with the new question as the body. To edit a question, the client will make a PUT request. The server will return a No Content (204) status with no body. To delete a question, the client will make a DELETE request and the server will again return a 204 status.

If a client makes a POST or PUT request and gives a question in the wrong model form, the server will return a Bad Request (409) status and a error message. If at any point in editing the list of questions we encounter an error, we return a Bad Request (409) status with an error message. If a PUT or DELETE request is made for a question that is not present, we return Not Found (404) with an error message. If a POST request is made and the client gives a preexisting question id, we return a Conflict (409) status and an error message. We will create an *enum* type of error messages to easily reference error messages in our controller code.

We will add another layer of abstraction to encapsulate the list of questions. This is called a repository layer. It will provide methods for operating on the list of questions. There will be a has-a relationship between the controller and the repository layer. The controller can then make simple calls to the repository layer, as opposed to manipulating the list itself.

3 Implementation

To implement our Web API, we will use the Microsoft ASP.NET Core framework. It is modern, open-source, modular, has an active community, provides an MVC framework, provides heavily featured templates, scales well due to easy multi-threading and deploys easily to Azure. Therefore, it is a good choice. We will develop using Visual Studio 2015.

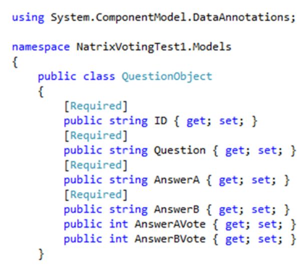
3.1 Creating a Project in Visual Studio

Navigate to File->New->Project. In the side bar, select Templates->Visual C#->.NET Core and select ASP.NET Core Web Application. Name the project, we have named it "NatrixVotingTest1". On the next screen, choose the Web API template and set authentication to "No Authentication".

3.2 Creating a Model for a Question

Create a new folder called Models. Within this, create a class called `QuestionObject`. See Figure 1. We mark the required fields with the 'Required' annotation from System.ComponentModel.DataAnnotations.

3.3 Creating the Repo Layer



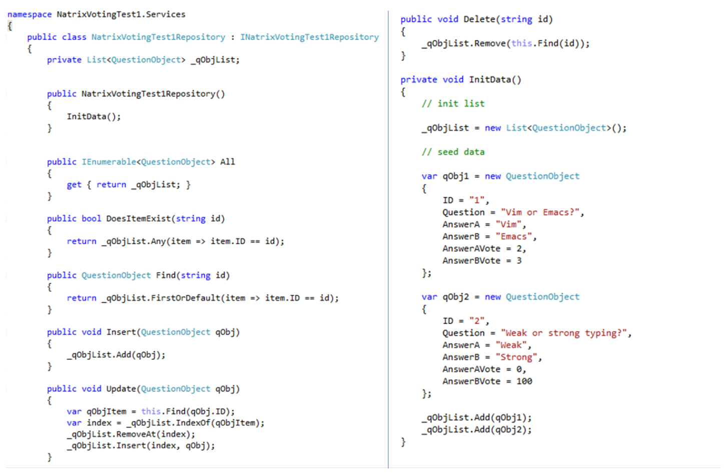
```
using System.ComponentModel.DataAnnotations;

namespace NatrixVotingTest1.Models
{
    public class QuestionObject
    {
        [Required]
        public string ID { get; set; }
        [Required]
        public string Question { get; set; }
        [Required]
        public string AnswerA { get; set; }
        [Required]
        public string AnswerB { get; set; }
        public int AnswerAVote { get; set; }
        public int AnswerBVote { get; set; }
    }
```

Figure 1: Model for a Question

We now want to create a layer than encapsulates the list of all `QuestionObjects`. First, we will create an interface, `INatrixTest1Repository`. It will specify methods for checking if an item exists given its `id`, returning the list of question items, finding a question by `id`, inserting a question, updating a question and deleting a question given an `id`. We will put this repo layer in `Interfaces/` folder.

We can now implement the repo layer. This is done using simple operations on a list. The implementations are given in Figure 2. The implementation class inherits the interface.

The list of questions is stored as a private attribute of this class. To access it, the `All` method acts as a getter. The constructor initialises this list, by calling the private method `InitData()`. This instantiates the list and 'seeds' it with two example test question objects, which are made by making instances of the `QuestionObject` class we made earlier. We should create the repo layer in the `services` folder.



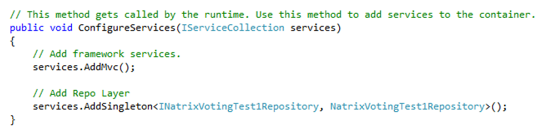
```
namespace NatrixVotingTest1.Services
{
    public class NatrixVotingTest1Repository : INatrixVotingTest1Repository
    {
        private List<QuestionObject> _qObjList;


        public NatrixVotingTest1Repository()
        {
            InitData();
        }


        public IEnumerable<QuestionObject> All
        {
            get { return _qObjList; }
        }

        public bool DoesItemExist(string id)
        {
            return _qObjList.Any(item => item.ID == id);
        }

        public QuestionObject Find(string id)
        {
            return _qObjList.FirstOrDefault(item => item.ID == id);
        }

        public void Insert(QuestionObject qObj)
        {
            _qObjList.Add(qObj);
        }

        public void Update(QuestionObject qObj)
        {
            var qObjItem = this.Find(qObj.ID);
            var index = _qObjList.IndexOf(qObjItem);
            _qObjList.RemoveAt(index);
            _qObjList.Insert(index, qObj);
        }
```

```
        public void Delete(string id)
        {
            _qObjList.Remove(this.Find(id));
        }

        private void InitData()
        {
            // init list

            _qObjList = new List<QuestionObject>();

            // seed data

            var qObj1 = new QuestionObject
            {
                ID = "1",
                Question = "Vim or Emacs?",
                AnswerA = "Vim",
                AnswerB = "Emacs",
                AnswerAVote = 2,
                AnswerBVote = 3
            };

            var qObj2 = new QuestionObject
            {
                ID = "2",
                Question = "Weak or strong typing?",
                AnswerA = "Weak",
                AnswerB = "Strong",
                AnswerAVote = 0,
                AnswerBVote = 100
            };

            _qObjList.Add(qObj1);
            _qObjList.Add(qObj2);
        }
```

Figure 2: Implementation for Repo Layer

Now we have created the repo layer, we can register it as a service. This means that, when we instantiate a constructor, for example, we can have it use ASP.NET Core dependency injection to request an instance of this repo layer. Therefore, when a controller gets instantiated, it gets an instance of the repo layer. To configure this, we need to add the repo to `services` in the `ConfigureServices` method of `Startup.cs`. The `AddSingleton` method call means that we only add the repository layer once: it's lifetime is the lifetime of the server. See Figure 3.

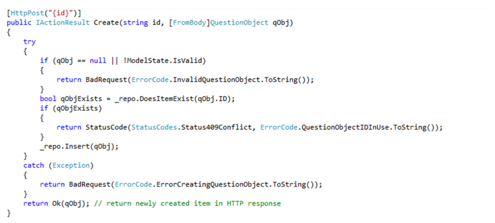
3.4 Creating the Controller



```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    // Add Repo Layer
    services.AddSingleton<INatrixVotingTest1Repository, NatrixVotingTest1Repository>();
}
```

Figure 3: Register repo layer with dependency injection

Create a new class `QuestionObjectsController` in the folder `Controllers/`. We need the imports shown in Figure 4. In the constructor, we add the repo layer as a parameter. ASP.NET Core dependency injection then gives us an instance of the repo layer when a controller is instantiated. We then assign it to a private class variable. Our controller methods can now access the question objects list via the methods of `_repo`. Also note that our controller class must inherit from the base controller class provided by ASP.NET. Lastly, we use a Route attribute to specify the route which gets routed to this controller. As we are only going to have up to one method for each HTTP verb GET, POST, PUT, DELETE, we only need one endpoint. The HTTP verb can be used to decide which controller action to take.



```
[HttpPost("{id}")]
public IActionResult Create(string id, [FromBody]QuestionObject qObj)
{
    try
    {
        if (qObj == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.InvalidQuestionObject.ToString());
        }
        bool qObjExists = _repo.DoesItemExist(qObj.ID);
        if (qObjExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.QuestionObjectIDInUse.ToString());
        }
        _repo.Insert(qObj);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.ErrorCreatingQuestionObject.ToString());
    }
    return Ok(qObj); // return newly created item in HTTP response
}
```

Figure 4: Controller Constructor

We can now implement the methods for each of the controller actions we described in Section 2: Design & Approach, with the correct semantics. In the same file, we also create an `enum ErrorCodes` to make it easier to maintain error messages for the different error cases.

The implementations for each of these methods are not included in this article for the sake of brevity. However, they may be viewed at the complimentary GitHub repository which contains the source code for this project.

We now have a working service for a mobile back-end. It provides an API with methods for all four CRUD operations. The GitHub repository is located at https://github.com/ahmerb/SampleWebAPI/.

4 Limitations & Alternatives

Recall, we made a key simplification when creating this back-end service. Namely, that we did not create a server that maintained some cloud-based database. This is a key requirement of almost all mobile applications that would be using a Web API for their mobile back-end. We justified this decision by stating that maintaining a database is a separate issue to creating an interface for mobile apps to communicate with. However, to evaluate whether this was a reasonable simplification, we need to question how hard it would be to add this feature.

Our controllers do not interface with the data directly. They interface with a repo layer. Therefore, to support a database, all we would do is create a new class that extends `INatrixTest1Repository` but that updates a database, as opposed to a list. The controller methods would not need to be changed. So, it is reasonably easy to add a database, and we have shown it to be a separate component to the Web API.

However, there are other limitations that one may point out. Firstly, in a production deployed application, we would expect a high level of verification via unit and integration testing. We have not demonstrated this. Testing is the only way to ensure the server does what it is expected. Therefore, it is crucial. Although not included here, the xUnit framework for ASP.NET Core applications makes it easy to write unit and integration tests. Therefore, if the solution presented here were to be adopted by a team in industry, they could indeed ensure they had a robust application.

With regards to alternative solutions, there are two categories of choice. Firstly, we can choose a different web development framework, such as Ruby on Rails. Different frameworks have different benefits, but the ASP.NET Core is a good choice. It scales easily. It is easy to create a multi-threaded back-end by using asynchronous controller actions. Deployment to Azure is easy, with which you can scale up you server sizes easily. As discussed, testing is also well supported. Secondly, we could choose to have an entirely differnt solution to provide mobile applications with a back-end service. This could entail having a raw database server to which the mobile app would connect to. This is not preferable for many reasons. For each mobile platform, you would have to redevelop the entire back-end. It is also less secure, as the entire back-end is now on the clients device. Also, the size of the consumer facing app would be significantly larger.

5 Conclusion

We have attempted to solve the problem of building a back-end service for mobile applications. Using reasonable simplifications, we have implemented a solution to this problem. We created a Web API using ASP.NET Core which exposes endpoints to which a mobile application can make HTTP requests to. The server is built using MVC and is easily scalable to a larger application. However, we have not deployed the application and showed it working with a mobile app. We have also not shown the application to be robust, by failing to give unit and integration tests. However, as we have discussed, the expansion of the implemented application to include these simplifications is straightforward. Therefore, we may conclude we have found a strong solution to creating back-end services for mobile applications.

6 Acknowledgements

The tutorial and walkthrough for creating a simple Web API is adapted from the tutorial by Steve Smith at
https://docs.microsoft.com/en-us/aspnet/core/mobile/native-mobile-backend

7 References

[1] Carney, Michael. "AnyPresence partners with Heroku to beef up its enterprise mBaaS offering". PandoDaily. Retrieved 24 June 2013.