第一章 入门

1. 建立工程
   dotnet new 模板名称 –o 输出文件夹

2. 查看可以建立什么模板
   dotnet new -l

3. 全局 using
   新建一个文件里面写 global using 命名空间

4. 类里面大写的成员才能被其他类访问

5. 一个简单的 property 写法
   public double Radius { get; set; }

6. string 类型的反转
   先用 ToCharArray()转 char[]，然后 Array.Reverse(arr)，再用 new string(arr)

7. 接口类用 I 开头

8. 可以同时集成多个类和接口
   public class Student : Person, IStudentAble

9. 对象初始化成员赋值直接用等号

```
var student = new Student() {
                Id = 1000,
                FirstName = "John",
                LastName = "Fry",
                Gender = "Male",
                School = "Business"
            };
```

10. 模板字符串用$
    $"{student.FirstName} {student.LastName} is studying {student.School}"

第二章 数据库

1. 创建项目后，需要先添加以下的包
   dotnet add package Microsoft.EntityFrameworkCore.Design
   dotnet add package Microsoft.EntityFrameworkCore.SqlServer
   dotnet add package Microsoft.EntityFrameworkCore.SqlServer.Design

2. 全局安装 dotnet-ef 用以创建 dbconext
   dotnet tool install --global dotnet-ef --version 6.0.1-*

3. 更新指定的 tool
   dotnet tool update --global dotnet-ef

4. 给当前程序添加 dbcontext
   dotnet-ef dbcontext scaffold "Data Source=max.bcit.ca;Database=Northwind;Persist Security Info=True;User ID=nw;Password=N0rthG@te;" Microsoft.EntityFrameworkCore.SqlServer -c NorthwindContext -o NW

5. 使用数据的步骤
   a. 初始化 context
      NorthwindContext db = new NorthwindContext();
   b. 使用
      foreach (var c in context.Categories) {
          Console.WriteLine($"{c.CategoryId}\t{c.CategoryName}\t{c.Description}");
      }

6. 查找数据的不同方式
   context.Categories.Where(c => c.CategoryName.StartsWith(starts));

   from c in context.Categories where c.CategoryName.StartsWith(starts) select c;

   context.Categories.Find(id);

7. 数据排序
   context.Categories.Select (c => new {c.CategoryId, c.CategoryName, c.Description }).OrderByDescending(c => c.CategoryId);

8. 选择部分数据的不同方式
   from c in context.Categories select new { c.CategoryId, c.CategoryName };

   context.Products.Select(p => new { p.ProductId, p.ProductName, p.Category.CategoryName });

9. 查询时更名数据
   context.Products.Select(p => new {ID = p.ProductId, Name = p.ProductName, Category = p.Category.CategoryName});

10. 数据分组统计
    context.Products
        .Include (c => c.Category)
        .GroupBy (p => p.Category.CategoryName)
        .Select (g => new { Name = g.Key, Count = g.Count () })
        .OrderByDescending(cp => cp.Count);

11. 添加记录

```
var newCategory = new Categories() {
    CategoryName = name,
    Description = desc
  };

 context.Categories.Add(newCategory);
 context.SaveChanges();
```

12. 更新数据

```
var categoryToUpdate = context.Categories.Find(id);
categoryToUpdate.CategoryName = name;
categoryToUpdate.Description = desc;
context.SaveChanges();
```

13. 执行原始的 SQL 指令

```
context.Database.ExecuteSqlRaw
```

14. 删除数据

```
var categoryToDelete = context.Categories.Find(id);
if (categoryToDelete != null)
{
   context.Categories.Remove(categoryToDelete);
   context.SaveChanges();
}
```

第三章 MVC

1. 分为 Controllers、Models、Views
2. Action 的返回值是 IActionResult
3. 给 View 传值可以有三种
   a. ViewBag 可以用[“”]和””
   b. ViewData 只能用[“”]
   c. 直接给 View 函数传值，页面用@model 解引用
4. Controller 对应的 View 在 Views/{ControllerName}/{ActionName}.cshtml
5. 共享的页面在 View/Shared 下
6. 静态资源在 wwwroot 下
7. 路由设置在 Program.cs 里

```
app.MapControllerRoute(name:"default",pattern:"{controller=Home}/{action=Index}
/{id?}");
```

8. 设置在 Properties 下的 launchSettings.json
9. 页面内的循环用@foreach(var i in Model)
10. 页面内可循环的变量@model IEnumerable<System.Diagnostics.Process>
11. 页面内的其他变量用@{}包裹和声明
12. 页面标题用 ViewData["Title"] = "Processes";

13. 页面变量类型用 as 声明
    System.Diagnostics.Process[] procs = ViewBag.Procs as System.Diagnostics.Process[];
14. 声明链接
    <a asp-controller="Process" asp-action="IndexPlus">Process Plus

    <a asp-controller="Process" asp-action="Display" asp-route-id="@i.Id">@i.ProcessName

第四章 Razor
1. 页面和代码全在 Pages 下
2. 每个页面 cshtml 对应一个 cshtml.cs
3. 页面头部单独声明
    @page
    @model ProcessModel //Model 名称
4. 声明链接
    Home
    <a asp-page="/Display" asp-route-id="@i.Id">@i.ProcessName</td>
5. 其他与 mvc 一致

第五章 MVC 和数据库集成
1. 安装工具
    dotnet tool install -g dotnet-aspnet-codegenerator
2. 创建 mvc 目录(不是项目)后进入到 mvc 目录初始化
    dotnet new mvc -f net6.0 --auth individual --use-local-db
3. 创建 Model
    dotnet-ef dbcontext scaffold "Data Source=max.bcit.ca;Database=Northwind;Persist Security Info=True;User ID=nw;Password=N0rthG@te;" Microsoft.EntityFrameworkCore.SqlServer -c NorthwindContext -o Models/NW
4. 安装包
    dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
5. 创建与模型对应的页面
    dotnet aspnet-codegenerator controller -name CategoriesController -outDir Controllers -m Category -dc NorthwindContext -udl –scripts
6. 添加 appsettings.json 里的连接字符串，删掉模型中的连接字符串
7. Program.cs 内添加数据库连接
    var nwConnectionString = builder.Configuration.GetConnectionString("NW");
    builder.Services.AddDbContext<NorthwindContext>(options => options.UseSqlServer(nwConnectionString));
8. 添加链接
9. ApplicationDbContext 用于用户验证
10. NorthwindContext 用于链接数据库
11. Controllers 里可以直接声明上述两种变量类型，直接被自动注入到构造函数的参数
12. 可以有[HttpGet]之类的 Method 注解

13. 异步请求 Action 声明
    public async Task<IActionResult> Details(int? id)
14. 异步取数据
    var category = await _context.Categories.FirstOrDefaultAsync(m => m.CategoryId == id);
15. Model 的 Annotation 命名空间 using System.ComponentModel.DataAnnotations;
    [Display(Name ="Category")] [MaxLength(15)] [MinLength(5)] [Required]

第六章 Docker

1. Docker 内容略
2. docker-compose.yaml 示例

```
version: '3.8'

services:
  db:
    image: mcr.microsoft.com/azure-sql-edge
    volumes:
      - sqlsystem:/var/opt/mssql/
      - sqldata:/var/opt/sqlserver/data
      - sqllog:/var/opt/sqlserver/log
      - sqlbackup:/var/opt/sqlserver/backup
    ports:
      - "1433:1433"
    restart: always
    environment:
      ACCEPT_EULA: Y
      MSSQL_SA_PASSWORD: SqlPassword!

  mvc:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - db
    ports:
      - "8888:80"
    restart: always
    environment:
      - DBHOST=db
      - DBPORT=1433
      - DBUSER=sa
      - DBPASSWORD=SqlPassword!
      - DBNAME=YellowDB
      - ASPNETCORE_ENVIRONMENT=Development
```

```
volumes:
  sqlsystem:
  sqldata:
  sqllog:
  sqlbackup:
```

3. C#读取环境变量

```
var host = builder.Configuration["DBHOST"] ?? "192.168.1.75";
var port = builder.Configuration["DBPORT"] ?? "1444";
var user = builder.Configuration["DBUSER"] ?? "sa";
var pwd = builder.Configuration["DBPASSWORD"] ?? "SqlPassword!";
var db = builder.Configuration["DBNAME"] ?? "YellowDB";
var conStr=$"Server=tcp:{host},{port};Database={db};UID={user};PWD={pwd};";
builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlS
erver(conStr));
```

第七章 SQLite 和 Migrate

1. 创建新项目时如果带了--auth individual 就是基于 SQLite 的，用户账户在里面

```
dotnet new mvc –auth individual -o Code1st
```

2. SQLite 数据库文件路径在 appSettings.json 里修改

3. 给 Model 的字段加上[key]标注，此字段即为主键

```
还需 using System.ComponentModel.DataAnnotations;
```

4. 一对多的声明

```
直接在 Model 类里加字段即可，例如 public List<Player>? Players { get; set; }
```

5. 多对一的声明

```
除了声明外键的 key 字段，例如 public string? TeamName { get; set; }
还要声明对应的对象：[ForeignKey("TeamName")] public Team? Team { get; set; }
```

6. 要在 context 里使用表，需要在 ApplicationDbContext 注册

```
public DbSet<Team>? Teams { get; set; }
public DbSet<Player>? Players { get; set; }
```

7. 添加测试数据，需要做两件事
   a. 加入测试数据代码

```
public class SampleData
    {
        public static List<Team> GetTeams()
        {
            List<Team> teams = new List<Team>() {
                new Team() {
                    TeamName="Lakers",
                    City="Los Angeles",
                },
            };

            return teams;
        }

        public static List<Player> GetPlayers()
        {
            List<Player> players = new List<Player>() {
                new Player {
                    PlayerId = 1,
                    FirstName = "LeBron",
                    LastName = "James",
                    TeamName = "Lakers",
                    Position = "Shooting Guard"
                },
            };
            return players;
        }
    }
```

   b. 在 ApplicationDbContext.cs 加入

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Team>().HasData(SampleData.GetTeams());
    modelBuilder.Entity<Player>().HasData(SampleData.GetPlayers());
}
```

8. **任何时候只要修改了 Model，都需要 Migrate，包括上述全部步骤**

```
dotnet ef migrations add M1 -o Data/Migrations //更新
dotnet ef database update //更新数据库，并加入测试数据
```

9. 添加自动生成 Controller 的包

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

10. 自动生成带 View 的 Controller

dotnet aspnet-codegenerator controller -name TeamsController -outDir Controllers -m Team -dc ApplicationDbContext -udl –scripts

dotnet aspnet-codegenerator controller -name PlayersController -outDir Controllers -m Player -dc ApplicationDbContext -udl –scripts

11. 自动生成不带 View 的 JSON Controller

dotnet aspnet-codegenerator controller -name TeamsController -outDir Controllers/api -m Team -dc ApplicationDbContext -actions -api

dotnet aspnet-codegenerator controller -name PlayersController -outDir Controllers/api -m Player -dc ApplicationDbContext -actions -api

12. 如果要在 JSON 结果中加入 Include，必须使用下面的方法
    a. dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
    b. 把下面的代码加入到 Program.cs 的 var app = builder.Build() 前面
       builder.Services.AddControllers().AddNewtonsoftJson(options =>
       options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore);

第八章 WebAPI 和跨域

1. 创建 WebAPI 项目，打开首页就能看到 swagger 生成的 API 列表

   dotnet new webapi -f net6.0 -o HealthAPI

2. WebAPI 的 Controller 继承自 ControllerBase 而不是 Controller
    a. 如果要提供 View 才需要继承自 Controller
    b. 还需要用[ApiController]和[Route(path/to/[controller])]来修饰

3. 如果没装 SQL 相关的全局包

   dotnet tool install --global dotnet-ef
   dotnet tool install -g dotnet-aspnet-codegenerator

4. 给项目安装 SQL 相关的包

   dotnet add package Microsoft.EntityFrameworkCore.Design
   dotnet add package Microsoft.EntityFrameworkCore.SqlServer.Design
   dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
   dotnet add package Microsoft.EntityFrameworkCore.SqlServer
   dotnet add package Microsoft.EntityFrameworkCore.Tools
   dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson

5. 用代码生成工具生成测试的 Controller

   dotnet aspnet-codegenerator controller -name ValuesController -async -api --readWriteActions -outDir Controllers

6. 在 appsettings.Development.json 里创建 SQL 连接字符串
   "ConnectionStrings": {"DefaultConnection": "Server=tcp:127.0.0.1,1444;Database=HealthDB;UID=sa;PWD=SqlPassword!;"},

7. 默认创建的项目里是没有 Data 和 Models 的，创建 Data 和 Models 目录
   创建相关的 Model 类，注意外键的写法
   ```
   [ForeignKey("PatientId")]
   public Patient? Patient { get; set; } //这个是外键的实体

   public int PatientId { get; set; } //这个是外键在此 Model 的 Column 名字
   ```

8. 创建 SampleData 类用以做初始化时填充数据
   参考上一章的 SampleData 部分

9. 在 Data 目录下新建 HealthContext.cs，用以把 Entity 和表绑定，并填充数据
   ```
   public class HealthContext :DbContext {
       public HealthContext(DbContextOptions options) : base(options) { }

       protected override void OnModelCreating(ModelBuilder builder) {
         base.OnModelCreating(builder);

         builder.Entity<Ailment>().Property(p =>p.Name).HasMaxLength(40);
         builder.Entity<Medication>().Property(p =>p.Name).HasMaxLength(40);
         builder.Entity<Patient>().Property(p =>p.Name).HasMaxLength(40);

         builder.Entity<Ailment>().ToTable("Ailment");
         builder.Entity<Medication>().ToTable("Medication");
         builder.Entity<Patient>().ToTable("Patient");

         builder.Entity<Patient>().HasData(SampleData.GetPatients());
         builder.Entity<Medication>().HasData(SampleData.GetMedication());
         builder.Entity<Ailment>().HasData(SampleData.GetAilments());
       }

       public DbSet<Ailment>? Ailments { get; set; }
       public DbSet<Medication>? Medications { get; set; }
       public DbSet<Patient>? Patients { get; set; }
   }
   ```

10. 把 9 中创建的 Context 在 Program.cs 里注册为服务，下面代码放 builder.Build()前
    ```
    var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
    builder.Services.AddDbContext<HealthContext>(options => options.UseSqlServer(connectionString));
    ```

11. 把下面的代码放在 10 里的代码之后

```
builder.Services.AddControllers().AddNewtonsoftJson(options =>
        options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.Reference
 LoopHandling.Ignore);
```

12. 执行 Migration 和 Update，来创建和更新数据库

```
dotnet ef migrations add M1 -o Data/Migrations
dotnet ef database update
```

13. 返回的数据里如果要加入外键的实体
    可以用 Include()指令，见前几章内容

14. 查询远程数据库需要用 async 相关的指令，如

```
return await _context.model.Include().ToListAsync();
```

15. 如果路由是带参数的，Get 函数应该做如下声明

```
[HttpGet("{id:int}/medication")]
public async Task<IActionResult> GetMedications(int id) {}
```

16. 客户端 jQuery 的 ajax 写法

```
$(function () {
  var getPatients = function () {
  var url = "https://localhost:5001/api/patients/";
  $.get(url).always(showResponse);
  return false;
};

var showResponse = function (object) {
      $("#preOutput").text(JSON.stringify(object, null, 4));
};

      $("#btnGetPatients").click(getPatients);
});
```

17. CORS 解决办法
    a. Program.cs 里在 Build()前加入

```
builder.Services.AddCors(o => o.AddPolicy("HealthPolicy", builder => {
  builder.AllowAnyOrigin()
    .AllowAnyMethod()
    .AllowAnyHeader();
}));
```

    b. UseAuthorization()前加入

```
app.UseRouting();
app.UseCors();
```

c. 用下面的注解标注 Controller 类

```
[EnableCors("HealthPolicy")]
```

18. PostMan 的使用

略

19. 部署后自动应用 Migration，把下面的代码放到 app.Run()前面

```
using (var scope = app.Services.CreateScope()) {
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<HealthContext>();
    context.Database.Migrate();
}
```

第九章 单元测试

1. C#的单元测试模块是 xunit

2. 新建类库的方法

```
dotnet new classlib -f net6.0
```

3. 新建测试工程的方法

```
dotnet new xunit -f net6.0
```

4. 常见的测试代码

Simple Equality
•Equal(1,2); // fail
•NotEqual(1,2); // pass
Ranges
•int v = 67;
•InRange(v, 21, 100); // pass
•NotInRange(v, 21, 100); // fail
Reference Equality
•Same(objA, objB); // fail
•NotSame(objA, objB) // pass
IEnumerable Emptiness
•Empty(new List<string>()); // pass
•string[] names = {"bob","sue"};
•NotEmpty(names); // pass

Boolean
•True(true); // pass
•False(true); // fail
Nulls
•Null(null); // pass
•NotNull("hello"); // fail
IEnumerable Contains Item
•Contains("May", months); // pass
•DoesNotContain("pen", months); // fail
SpecificType
•IsType<string>("hello"); // pass
•IsNotType<int>("hello"); // pass
Exception
•var input = "a string";
•Assert.Throws<FormatException>(() => int.Parse(input));

5. 运行测试

```
dotnet test
```

6. 工程添加对其他工程的引用
   a. dotnet add reference 相对路径
   b. 也可以编辑该项目的配置文件

```
<ItemGroup>
    <ProjectReference Include="..\FizzBuzzLibrary\FizzBuzzLibrary.csproj" />
</ItemGroup>
```

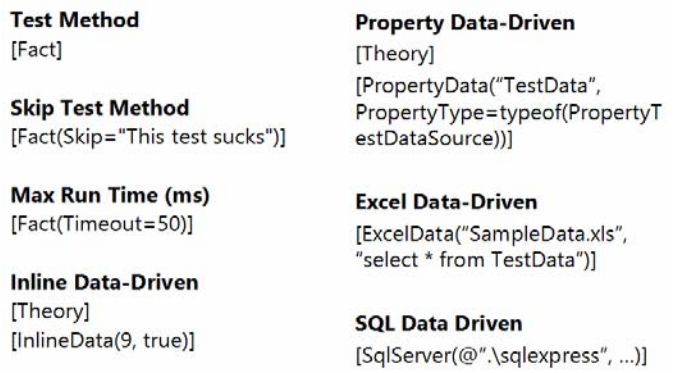
7. 测试流程

   Red: 创造能导致失败的测试代码

   Greed: 写仅仅保证能通过测试的代码

   Refactor: 清理代码，但不导致测试失败

8. 常见的测试注解



第十章 集成 MongoDB 数据库

1. 启动一个 mogodb 的容器

   docker run -p 27777:27017 --name mgo -d mongo:4.1.6

2. Mongodb 常用指令

   use dbname

```
db.Students.insertMany(
[
  {'FirstName':'Sue','LastName':'Fox','School':'Business'},
  {'FirstName':'Tom','LastName':'Max','School':'Mining'},
]);

db.Students.find({}).pretty();

exit
```

3. 给项目增加对 MongoDB 的支持

   dotnet add package MongoDB.Driver --version 2.14.1

4. Mongodb 的 Model 长这样，注意红色部分

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

public class Student {
    [BsonId]  //必须要，做主键
    [BsonRepresentation(BsonType.ObjectId)] //允许传入字符串而不用 ObjectID
    public string? Id { get; set; }
```

```
    public string? FirstName { get; set; }
    public string? LastName { get; set; }

    [BsonElement("School")] //当变量名字和表 field 名字不同时使用
    public string? Department { get; set; }
}
```

5. 对项目设置的修改

```
"StudentDbSettings": {
 "CollectionName": "Students",
 "ConnectionString": "mongodb://localhost:27777",
 "DatabaseName": "school-db"
},
```

6. Models 里面再加一个

```
public class StudentsDbSettings {
    public string ConnectionString { get; set; } = null!;
    public string DatabaseName { get; set; } = null!;
    public string CollectionName { get; set; } = null!;
}
```

7. 主程序代码里的 Build()之前加入

```
builder.Services.Configure<StudentsDbSettings>(
    builder.Configuration.GetSection("StudentDbSettings"));
```

8. 创建一个 Service，用于注入增删改查功能

```
public class StudentsService {
    private readonly IMongoCollection<Student> _studentsCollection;

    public StudentsService(IOptions<StudentsDbSettings> studentsDatabaseSetting
s) {
        var mongoClient = new MongoClient(
            studentsDatabaseSettings.Value.ConnectionString);

        var mongoDatabase = mongoClient.GetDatabase(
            studentsDatabaseSettings.Value.DatabaseName);

        _studentsCollection = mongoDatabase.GetCollection<Student>(
            studentsDatabaseSettings.Value.CollectionName);
    }

    public async Task<List<Student>> GetAsync() =>
        await _studentsCollection.Find(_ => true).ToListAsync();
```

```
    public async Task<Student?> GetAsync(string id) =>
        await _studentsCollection.Find(x => x.Id == id).FirstOrDefaultAsync();

    public async Task CreateAsync(Student newStudent) =>
        await _studentsCollection.InsertOneAsync(newStudent);

    public async Task UpdateAsync(string id, Student updatedStudent) =>
        await _studentsCollection.ReplaceOneAsync(x => x.Id == id, updatedStu
dent);

    public async Task RemoveAsync(string id) =>
        await _studentsCollection.DeleteOneAsync(x => x.Id == id);
}
```

9. 在主程序代码里面的 Build()之前加上
   builder.Services.AddSingleton<StudentsService>();

10. 在 Controller 里就可以注入使用 Service 了
```
private readonly StudentsService _studentsService;

public StudentsController(StudentsService studentsService) =>
    _studentsService = studentsService;
```

第十一章　Blazor & Ajax

1. Blazor 其实就是用 C#来生成 Ajax 相关的代码和页面

2. 创建一个 Blazor 工程
   dotnet new blazorwasm -o BlazorClient

3. 使用命名空间应该在**_Import.razor** 里添加，才能被页面使用
   @using StudentsLibrary

4. 页面的路由地址在页面的 razor 文件里规定
```
@page "/students" //定义页面的路由地址
@inject HttpClient httpClient //注入 http 客户端用以请求 JSON
```

5. 页面里面可以直接写 HTML 代码，页面标题用<PageTitle>Title</PageTitle>定义

6. 条件渲染的实现
```
@if (students == null) {
  @code{     //这里还能写代码，还能引用全局@code 里的变量
  }
} else {
}
```

7. 全局@code 里的 protected override async Task OnInitializedAsync()在页面加载时执行，一般用于初始化页面

8. 侧边栏的菜单在 Shared/NavMenu.razor 里面改

9. 如何提交一个表单

```
@if (students != null && mode == MODE.Add) // Insert form
{
  <EditForm Model="@student" OnValidSubmit="@HandleAdd">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText placeholder="First Name" id="firstName" @bind-Value="@student.
FirstName" />
    <InputText placeholder="Last Name" id="lastName" @bind-Value="@student.
LastName" />
    <InputText placeholder="School" id="school" @bind-Value="@student.School"
 />
    <button type="submit">Submit</button>
  </EditForm>

    @code {
      private Student student = new Student();
      private async void HandleAdd()  {
        string endpoint = $"{baseUrl}/api/students";
        student.StudentId = Guid.NewGuid().ToString();
        await httpClient.PostAsJsonAsync(endpoint, student);
        mode = MODE.None;
        await load();
        StateHasChanged(); // causes the page to get automatically refreshed
      }
   }
}
```

10. 元素绑定事件

调用不带参数的函数

```
<button @onclick="@Add">Add</button>
@code 里
protected void Add() {
  mode = MODE.Add;
}
```

调用带参数的函数

```
<a @onclick="@(() => ShowEdit(item.StudentId))">edit</a></td>
```

11. HttpClient 的几个主要方法

GetFromJsonAsync<>

DeleteAsync

PutAsJsonAsync

PostAsJsonAsync

第十二章 WebSocker 的 SignalR 实现

1. 创建项目

```
dotnet new webapp --no-https -o SignalrChat
```

2. 安装 LibMan (微软包管理工具)

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

3. 用 LibMan 安装 SignalR，文件 libman.json 出现在项目根目录下(类似 package.json)

```
libman install @aspnet/signalr -p unpkg -d wwwroot/lib/signalr --files dist/browse
r/signalr.js --files dist/browser/signalr.min.js
```

4. 新建一个 Hub 目录，在下面新建一个 ChatHub 类

```
using Microsoft.AspNetCore.SignalR;
public class ChatHub : Hub {
  public async Task SendMessage(string user, string message) {
    await Clients.All.SendAsync("ReceiveMessage", user, message);
  }
}
```

5. 主程序 Build()之前加上

```
builder.Services.AddSignalR();
```

6. 主程序 app.MapRazorPages()后面加上

```
app.MapHub<ChatHub>("/chatHub");
```

7. Pages/Index.cshtml 改为（代码里的~代表 wwwroot 目录）

```
@page
<input type="text" id="userInput" />
<input type="text" id="messageInput" />
<input type="button" id="sendButton" value="Send Message" />
<ul id="messagesList"></ul>
<script src="~/lib/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

8. ~/js/chat.js 的代码

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable send button until connection is established
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "
&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().then(function(){
    document.getElementById("sendButton").disabled = false;
    }).catch(function (err) {
        return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event)
{
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
            return console.error(err.toString());
        });
    event.preventDefault();
});
```

第十三章 Blazor 服务端渲染以及 MySQL 连接

1. 创建项目

```
dotnet new blazorserver -o ServerBlazorEF
```

2. 添加依赖包

```
dotnet add package Pomelo.EntityFrameworkCore.MySql -v 6.0.1
dotnet add package Microsoft.EntityFrameworkCore.Tools -v 6.0.1
dotnet add package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore  -v 6.
0.1
```

3. 启动 Docker 里的 MySQL

```
docker run -p 3333:3306 --name mydb -e MYSQL_ROOT_PASSWORD=secret -d mysql:8.0.0
```

4. 添加 ConnectionString

```
"ConnectionStrings": {
    "DefaultConnection": "server=localhost; userid=root; pwd=secret; port=3333; database=school; SslMode=none;"
},
```

5. 创建模型

```
public class Student {
    public string?StudentId{ get; set; }
    [Required]
    public string? FirstName { get; set; }
    [Required]
    public string?LastName{ get; set; }
    [Required]
    public string? School { get; set; }
}
```

6. 创建 Context

```
public class SchoolDbContext: DbContext {
    public DbSet<Student>? Students { get; set; }

    public SchoolDbContext(DbContextOptions<SchoolDbContext> options) : base (options) { }

    protected override void OnModelCreating(ModelBuilder builder) {
base.OnModelCreating(builder);

builder.Entity<Student>().HasData(
        new {
            StudentId = Guid.NewGuid().ToString(),
            FirstName = "Jane",
            LastName = "Smith",
          School = "Medicine"
        });
    }
}
```

7. Program.cs 添加 Context

```
var serverVersion = new MySqlServerVersion(new Version(8, 0, 0));
var connectionString = builder.Configuration.GetConnectionString("DefaultConnectio
```

```
n");
builder.Services.AddDbContext<SchoolDbContext>(option => option.UseMySql(conn
ectionString,serverVersion));
```

8. 创建 Service 用于提供增删改查的接口

```
public class StudentService
    {
        SchoolDbContext _context;
        public StudentService(SchoolDbContext context)
        {
            _context = context;
        }

        public async Task<List<Student>> GetStudentsAsync()
        {
            var result = _context!.Students;
            return await Task.FromResult(result!.ToList());
        }

        public async Task<Student> GetStudentByIdAsync(string id)
        {
            return await _context.Students!.FindAsync(id);
        }

        public async Task<Student> InsertStudentAsync(Student student)
        {
            _context.Students!.Add(student);
            await _context.SaveChangesAsync();

            return student;
        }

        public async Task<Student> UpdateStudentAsync(string id, Student s)
        {
            var student = await _context.Students!.FindAsync(id);

            if (student == null)
                return null!;

            student.FirstName = s.FirstName;
            student.LastName = s.LastName;
            student.School = s.School;

            _context.Students.Update(student);
```

```
            await _context.SaveChangesAsync();

            return student;
        }

        public async Task<Student> DeleteStudentAsync(string id)
        {
            var student = await _context.Students!.FindAsync(id);

            if (student == null)
                return null!;

            _context.Students.Remove(student);
            await _context.SaveChangesAsync();

            return student;
        }

        private bool StudentExists(string id)
        {
            return _context.Students!.Any(e => e.StudentId == id);
        }
    }
```

9. Program.cs 內加入 service

```
builder.Services.AddScoped<StudentService>();
```

10. Pages 內加入新页面

```
@page "/students"
@using ServerBlazorEF.Data
@using ServerBlazorEF.Models
@inject StudentService studentService
<h1>Students</h1>

<p>This component demonstrates managing students data.</p>

@if (students == null) {
<p><em>Loading...</em></p>
} else {
<table class='table table-hover'>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
```

```
<th>Last Name</th>
<th>School</th>
</tr>
</thead>
<tbody>
 @foreach (var item in students)
 {
<tr>
<td>@item.StudentId</td>
<td>@item.FirstName</td>
<td>@item.LastName</td>
<td>@item.School</td>
</tr>
}
</tbody>
</table>
}


@code {
  List<Student>?students;

  protected override async Task OnInitializedAsync() {
    await load();
  }

  protected async Task load() {
    students = await studentService.GetStudentsAsync();
  }
}
```

11. 页面的写法基本与 Blazor 一致，只是注入的不是 httpclient，而是服务

第十四章 React

1. BrowserRouter 要把组件写进去，否则<Link>无法使用

```
<BrowserRouter>
  <NavMenu />
  <Routes>
    <Route path="/" element={<Home />} exact />
    <Route path="/Privacy" element={<Privacy />} exact />
  </Routes>
  <Footer />
</BrowserRouter>
```

2. Router 里面加参数的办法
   <Route path="/detail/:id" element={<ToonDetailPage />} exact />
   组件内 const { id } = useParams();

第十五章 Azure 函数应用

1. 安装 Azure Functions Core Tools
2. 安装 VS Code 的 Azure Function 扩展
3. 在 Azure Function 的 Tab 内创建 Function 代码
   C# -> HttpTrigger -> Anonymous
4. func run 或者 VS Code 的 Ctrl+F5 可以运行此项目
5. 为了连接 MSSQL 数据库，安装如下包
   dotnet add package Microsoft.Azure.Functions.Extensions
   dotnet add package Microsoft.NET.Sdk.Functions
   dotnet add package Microsoft.EntityFrameworkCore
   dotnet add package Microsoft.EntityFrameworkCore.Design
   dotnet add package Microsoft.EntityFrameworkCore.SqlServer
   dotnet add package Microsoft.EntityFrameworkCore.Tools
6. 去掉 Controller 类的 static 修饰符
7. 创建 Models
   dotnet-ef dbcontext scaffold "Server=max.bcit.ca;UID=flintstone;Password=Fl1nt$t0ne;Database=Toons;" Microsoft.EntityFrameworkCore.SqlServer -c ToonsContext -o Models\Toons
8. local.settings.json 内加入连接字符串(部署 Azure 后是设置环境变量)
   "ConnectionStrings": {
   " Toons": "Server=max.bcit.ca;UID=flintstone;Password=Fl1nt$t0ne;Database=Toons;"
   },
9. 删除 OnConfiguring()函数
10. 创建 Startup.cs 来注入 context

```
using System;
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

[assembly: FunctionsStartup(typeof(Snoopy.Function.StartUp))]
namespace Snoopy.Function
{
    public class StartUp : FunctionsStartup {
        public override void Configure(IFunctionsHostBuilder builder) {
            string connStr = Environment.GetEnvironmentVariable("ConnectionStrings:Toons");

            builder.Services.AddDbContext<ToonsContext>(
              options => SqlServerDbContextOptionsExtensions.UseSqlServer(opti
```

```
ons, connStr));
            }
        }
    }
```

11. Controller 类加入 context

```
private readonly ToonsContext _context;

public HttpWebAPI(ToonsContext context) {
    _context = context;
}
```

12. Controller 类加入新的 Route 地址

```
[FunctionName("GetToons")]
public IActionResult GetToons(
  [HttpTrigger(AuthorizationLevel.Function, "get", Route = "toons")] HttpRequest r
eq,
  ILogger log)
{
  log.LogInformation("C# HTTP GET/posts trigger function processed a request i
n GetToons().");

  var toons = _context.People.ToArray(); //要导入 linq

  return new OkObjectResult(toons);
}
```

13. 发布到 Azure 略

第十六章 国际化

1. 这里主要讲的是 mvc 的国际化，与其他模块无关

2. 建立一个 Resources 目录放翻译的字符串资源

3. 在 Program.cs 里加入或替换如下代码
   a. 在 builder.build()之前

```
builder.Services.AddControllersWithViews()
.AddDataAnnotationsLocalization(
    option =>
    {
        option.DataAnnotationLocalizerProvider = (type, factory) =>
        {
            var asmName = new AssemblyName(typeof(AnnoData).GetTypeI
nfo().Assembly.FullName!);
            return factory.Create("DataAnno", asmName.Name!);
        };
    }
```

```
)
.AddViewLocalization();

builder.Services.AddLocalization(options => options.ResourcesPath = "Resources");
```

b. 在 builder.Build()之后

```
var supportedCultures = new[] {
    "en","en-US","en-CA","fr","fr-FR","zh-CN","ar","ar-EG","ko-KR","ru-RU"
};

var localizationOptions = new RequestLocalizationOptions().SetDefaultCulture(supportedCultures[1])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

localizationOptions.ApplyCurrentCultureToResponseHeaders = true;

app.UseRequestLocalization(localizationOptions);
```

4. 基于 Controller 的翻译
   a. 在 Resources 目录下建立与 Controller 对应的 resx 文件，放翻译字符串，例如 HomeController.fr.resx
   b. Controller 内注入 `private readonlyIStringLocalizer<HomeController> _localizer;`
   c. 需要翻译的字符串用 ViewData 推到页面，如:

```
ViewData["pressRelease"] = _localizer["Press Release"];
ViewData["welcome"] = _localizer.GetString("Welcome").Value ?? "";
```

5. 多页面共享翻译资源
   a. 在 Resources 目录下建立固定名称规则的资源文件 SharedResource.lang.resx
   b. Controller 内注入 `private readonlyIStringLocalizer<SharedResource> _sharedLocalizer;`
   c. 需要翻译的字符串用 ViewData 传递，如：

```
ViewData["pressRelease"] = _sharedLocalizer["Press Release"];
ViewData["welcome"] = _sharedLocalizer.GetString("Welcome").Value ?? ""
```

6. 基于 View 的翻译
   a. 在 Resources 目录下建立与 View 文件对应的 resx 文件，如: Resources/Views/Home/Privacy.fr.resx
   b. View 文件顶部放上

```
@using Microsoft.AspNetCore.Mvc.Localization
@injectIViewLocalizer _localizer
```

   c. View 内部直接使用翻译资源，如：

```
ViewData["Title"] = _localizer["Privacy Policy"];
```

7. 基于 Model 的翻译
   a. 直接给 Model 属性加上注解，例如
      [Display(Name = "Your Email")]
      public string? Email { get; set; }
   b. 在 Resources 目录下建立与 Model 文件对应的 resx 文件，如: Resources/Models/Contact.fr.resx (如果用我自己的代码，直接在 Resouces 根目录下建 DataAnno.fr.resx 即可)
   c. 页面代码可用 dotnet 代码生成工具生成，无需修改内容，具体可看第五章第 5 条

第十七章 给 MVC 的用户验证添加更多字段

1. 创建带有用户验证的 MVC 程序
   dotnet new mvc --auth individual -f net6.0 -o IdentityCore

2. 新建两个 Models：CustomUser 和 CustomRole，用于新增字段

```
public class CustomUser :IdentityUser {
  public CustomUser() : base() { }

  public string FirstName { get; set; }
  public string LastName{ get; set; }
}
```

和

```
public class CustomRole :IdentityRole {

  public CustomRole() : base() { }

  public CustomRole(string roleName) : base(roleName) { }

  public CustomRole(string roleName, string description,
DateTimecreatedDate): base(roleName) {
    base.Name = roleName;

    this.Description = description;
    this.CreatedDate = createdDate;
  }

  public string Description { get; set; }
  public DateTimeCreatedDate{ get; set; }
}
```

3. 修改 ApplicationAbContext 为 ApplicationDbContext :IdentityDbContext<CustomUser, CustomRole, string>

4. 新增 IdentitySeedData 用于填充 seed 数据，见下页

```
public class IdentitySeedData {
    public static async Task Initialize(ApplicationDbContext context,
      UserManager<CustomUser>userManager,
      RoleManager<CustomRole>roleManager)
  {
      context.Database.EnsureCreated();

      string asdminRole = "Admin";
      string adminDesc = "This is the administrator role";

      string memberRole = "Member";
      string memberDesc = "This is the members role";

      string password4all = "P@$$w0rd";

      if (await roleManager.FindByNameAsync(asdminRole) == null) {
        await roleManager.CreateAsync(new CustomRole(asdminRole, adminDesc, Dat
eTime.Now));
      }

      if (await roleManager.FindByNameAsync(memberRole) == null) {
        await roleManager.CreateAsync(new CustomRole(memberRole, memberDesc,
DateTime.Now));
      }

      if (await userManager.FindByNameAsync("aa@aa.aa") == null) {
          var user = new CustomUser {
            UserName = "aa@aa.aa",
            Email = "aa@aa.aa",
            FirstName = "Adam",
            LastName = "Aldridge",
            PhoneNumber = "6902341234"
          };

          var result = await userManager.CreateAsync(user);
          if (result.Succeeded) {
            await userManager.AddPasswordAsync(user, password4all);
            await userManager.AddToRoleAsync(user, asdminRole);
          }
      }
  }
}
```

5. 修改 Program.cs 里的 AddDefaultIdentity 为

```
builder.Services.AddIdentity<CustomUser, CustomRole>(
options => {
    options.Stores.MaxLengthForKeys = 128;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddRoles<CustomRole>()
.AddDefaultUI()
.AddDefaultTokenProviders();
```

6. 修改 Views/Shared/_LoginPartial.cshtml 的注入

```
@injectSignInManager<CustomUser>SignInManager
@injectUserManager<CustomUser>UserManager
```

7. 在 Program.cs 的 Run()之前加上自动合并和自动 Seed

```
using (var scope = app.Services.CreateScope()) {
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ApplicationDbContext>();
    var roleManager = services.GetRequiredService<RoleManager<CustomRole>>
();
    var userManager = services.GetRequiredService<UserManager<CustomUser>>
();
     IdentitySeedData.Initialize(context, userManager, roleManager).Wait();
}
```

8. 生成 migration 并更新数据库

```
dotnet-ef migrations add M1 -o Data/Migrations
dotnet-ef database update
```

9. 增加包

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

10. 暴露 login 的相关页面，方便修改

```
dotnet aspnet-codegenerator identity --files "Account.Register;Account.Login;Accou
nt.RegisterConfirmation"
```

这里有个坑，它会修改 Program.cs，导致 ApplicationDbContext 和之前修改的 AddDef aultIdentity 变成同一行，删除的时候容易误删，导致程序无法启动，要特别注意

11. 修改 Areas\Identity\Pages\Account\Register.cshtml.cs，在 InputModel 里加上

```
[Required]
[DataType(DataType.Text)]
[StringLength(50, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 2)]
[Display(Name ="First Name")]
public string FirstName { get; set; }

[Required]
[DataType(DataType.Text)]
[StringLength(50, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 2)]
[Display(Name = "Last Name")]
public string LastName{ get; set; }
```

在 OnPostAsync()函数的 user = CreateUser()下面加

```
user.FirstName = Input.FirstName;
user.LastName = Input.LastName;
```

12. 修改 Areas\Identity\Pages\Account\Register.cshtml，加 LastName 和 FirstName 的输入

```
<div class="form-floating">
<input asp-for="Input.FirstName" class="form-control" autocomplete="firstname" aria-required="true" />
<label asp-for="Input.FirstName"></label>
<span asp-validation-for="Input.FirstName" class="text-danger"></span>
</div>

<div class="form-floating">
<input asp-for="Input.LastName" class="form-control" autocomplete="lastname" aria-required="true" />
<label asp-for="Input.LastName"></label>
<span asp-validation-for="Input.LastName" class="text-danger"></span>
</div>
```

13. 修改 Register.cshtml.cs 和 Login.cshtml.cs，把 IdentityUser 批量替换为 CustomUser

14. 删掉 Program.cs 里面的如下内容和相关 using

```
builder.Services.AddDbContext<IdentityCoreIdentityDbContext>(options =>
    options.UseSqlServer(connectionString));

builder.Services.AddDefaultIdentity<IdentityUser>(options =>options.SignIn.RequireConfirmedAccount = true).AddEntityFrameworkStores<IdentityCoreIdentityDbContext>();
```

第十八章 JWT 和 Token 的实现

1. 创建 Webapi 项目

```
dotnet new webapi-f net6.0 -o TokenAuth
```

2. 安装 JWT 相关的包

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package Microsoft.AspNetCore.Identity
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SQLite
dotnet add package Microsoft.EntityFrameworkCore.SQLite.Design
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package System.IdentityModel.Tokens.Jwt
```

3. 在 Data 目录下新增 ApplicationDbContext 类

```
public class ApplicationDbContext:IdentityDbContext<IdentityUser> {
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
 : base(options) {}

    protected override void OnModelCreating(ModelBuilder builder) {
base.OnModelCreating(builder);

    #region "Seed Data"

    builder.Entity<IdentityRole>().HasData(
            new {Id = "1", Name = "Admin", NormalizedName = "ADMIN"},
            new { Id = "2", Name = "Customer", NormalizedName = "CUSTOM
      ER" }
    );

    #endregion
  }
}
```

4. 在 ViewModels 下新增 LoginViewModel.cs 和 RegisterViewModel.cs

```
public class LoginViewModel {
  public string? Username { get; set; }
  public string? Password { get; set; }
}
```

和

```
public class RegisterViewModel {
  [Required]
  [EmailAddress]
  public string Email?{ get; set; }
```

```
    [Required]
    public string? Password { get; set; }
}
```

5. 在 appsettings.json 里新增数据库连接配置和 JWT 的配置

```
"ConnectionStrings": {
      "DefaultConnection": "DataSource=app.db"
},
"Jwt": {
  "Site": "http://www.security.org",
  "SigningKey": "Paris Berlin Cairo Sydney Tokyo Beijing Rome London Athens",
  "ExpiryInMinutes":  "60"
},
```

6. 在 Program.cs 的 var app = builder.Build()前面新增

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnectio
n");
builder.Services.AddDbContext<ApplicationDbContext>(
 option =>option.UseSqlite(connectionString));

builder.Services.AddIdentity<IdentityUser, IdentityRole>(
 option =>
 {
      option.Password.RequireDigit = false;
      option.Password.RequiredLength = 6;
      option.Password.RequireNonAlphanumeric = false;
      option.Password.RequireUppercase = false;
      option.Password.RequireLowercase = false;
 }
).AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();
```

7. 新增一个名为 AuthController 的 Controller，负责用户注册和登录，登录时返回 token

```
public class AuthController : Controller
{
     private readonlyUserManager<IdentityUser> _userManager;
     private readonlyIConfiguration _configuration;

     public AuthController(UserManager<IdentityUser>userManager, IConfiguration
configuration)
     {
          _userManager = userManager;
          _configuration = configuration;
     }
```

```
    [Route("register")]
    [HttpPost]
    public async Task<ActionResult>InsertUser([FromBody] RegisterViewModel mo
del)
    {
        var user = new IdentityUser
        {
            Email = model.Email,
UserName = model.Email,
SecurityStamp = Guid.NewGuid().ToString()
        };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await _userManager.AddToRoleAsync(user, "Customer");
        }
        return Ok(new { Username = user.UserName });
    }

    [Route("login")]
    [HttpPost]
    public async Task<ActionResult> Login([FromBody] LoginViewModel model)
    {
        var user = await _userManager.FindByNameAsync(model.Username);
        if (user != null && await _userManager.CheckPasswordAsync(user, model.
Password))
        {
            var claim = new[] {
    new Claim(JwtRegisteredClaimNames.Sub, user.UserName)
  };
            var signinKey = new SymmetricSecurityKey(
              Encoding.UTF8.GetBytes(_configuration["Jwt:SigningKey"]));

            int expiryInMinutes = Convert.ToInt32(_configuration["Jwt:ExpiryInMi
nutes"]);

            var token = new JwtSecurityToken(
              issuer: _configuration["Jwt:Site"],
              audience: _configuration["Jwt:Site"],
              expires: DateTime.UtcNow.AddMinutes(expiryInMinutes),
signingCredentials: new SigningCredentials(signinKey, SecurityAlgorithms.HmacSha2
56)
            );
```

```
                return Ok(
                    new
                    {
                        token = new JwtSecurityTokenHandler().WriteToken(token),
                        expiration = token.ValidTo
                    });
            }
            return Unauthorized();
        }
    }
```

8. 修改 Program.cs，加上

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

并在 var app = builder.Build()后面加上

```
builder.Services.AddAuthentication(option => {
option.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
option.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
option.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options => {
options.SaveToken = true;
options.RequireHttpsMetadata = true;
options.TokenValidationParameters = new TokenValidationParameters()
  {
ValidateIssuer = true,
ValidateAudience = true,
ValidAudience = builder.Configuration["Jwt:Site"],
ValidIssuer = builder.Configuration["Jwt:Site"],
IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Con
figuration["Jwt:SigningKey"]))
  };
});
```

9. 修改 Program.cs 开启 token 认证功能

```
app.UseAuthentication();
```

10. 创建 migration 并更新数据库(略)

11. 在任何想要用 token 限制用户的类上面加[Authorize]注解
    无 token 时，访问其内部接口就会返回 401 错误

12. 可用 PostMan 或 jQuery 测试，拿着 token 去 jwt.io 解码看内容
    有需要时看 Medhad 的 Script 怎么操作

第十九章 gRPC

1. dotnet 的 gRPC 必须在 Win10 才支持，具体名词定义看教学脚本

2. 新建 gRPC 服务端
   dotnet new gRPC -o gRPCServer

3. 例子 proto 文件如下，所有的 proto 文件都会在 build 时自动编译为 cs 文件

```
syntax = "proto3";

option csharp_namespace = "gRPCServer"; //命名空间

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply); //接口函数，前者参数后者返回
}

// The request message containing the user's name.  //定义的参数或返回值结构
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

4. proto 一般文件放在 Protos 文件夹下，并在 csproj 文件里增加引用如下
   <Protobuf Include="Protos\greet.proto" GrpcServices="Server" /> //服务端

5. 服务端实现 proto 文件里面的接口，使用下面这样的代码(放在 Services 目录下)

```
public class GreeterService : Greeter.GreeterBase //基类是自动生成的
{
    //构造函数略，实现函数要与 proto 文件里的声明对应
    public override Task<HelloReply> SayHello(HelloRequest request, ServerCallCo
ntext context){
        return Task.FromResult(new HelloReply {
            Message = "Hello " + request.Name
        });
    }
}
```

6. 服务端安装这些包来支持 SQLite 数据库，支持数据库步骤与前文一致，因此略

```
dotnet add package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore -v 6.0
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore -v 6.0
dotnet add package Microsoft.EntityFrameworkCore.Sqlite -v 6.0
dotnet add package Microsoft.EntityFrameworkCore.Design -v 6.0
dotnet add package Grpc.Tools
```

7. 新增的 Service 要在 Program.cs 里进行注册，在 var app = builder.Build();下面加
   app.MapGrpcService<StudentsService>();

8. 客户端也要包含与服务端一样的 proto 文件，也要在工程里添加 ItemGroup
   <Protobuf Include="Protos\greet.proto" GrpcServices="Client" /> //客户端

9. 客户端用如下的代码调用服务器函数

```
var input = new HelloRequest { Name = "Jane Bond" };
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(input);
Console.WriteLine(reply.Message);
```

10. 在 proto 里定义数组，用如下的代码

```
message StudentModel {
  int32 studentId = 1;
  string firstName = 2;
  string lastName = 3;
  string school = 4;
}

message StudentList { repeated StudentModel items = 1; }
```

11. proto 还会自动生成带 Async 的异步函数，根据需要调用

12. proto 的 service 函数必须要有参数，所以可以定义一个空 message 作为参数

```
service StudentRemote {
  rpc RetrieveAllStudents(Empty) returns (StudentList);
}

message Empty {}
```

13. 详细的代码: https://github.com/Jobcrazy/BCIT/tree/master/C%23/Lab13