Challenge 2

By: Mosh Hamedani

Building APIs for CRUD Operations

Our form is ready. Soon, we need to post this form to the server to create or update vehicles. So, as part of this challenge, you should create a new API for creating, updating, deleting and getting **Vehicle** objects. This API should be exposed at **/api/vehicles.**

Test these endpoints using Postman Chrome plugin. Ensure that all actions return the proper responses and support data validation. Use data annotations to implement validation. The following properties of a vehicle are required:

- Model
- ContactName (max 255 characters)
- ContactPhone (max 255 characters)

Also, each Vehicle should have a **LastUpdate** property, which should be set at the time it is created or updated.

API Responses

In ASP.NET MVC5, depending on the type of our controllers, our actions have different return types:

- **MVC controllers:** an instance of a class that derives from **ActionResult**
- **API controllers:** an instance of a class that implements **IHttpActionResult**

In ASP.NET Core, our controllers are consolidated into one Controller class. Our actions now can return **IActionResult**. There are different classes that implement this interface. With these, we can return both JSON objects as well as Razor views.

Here are some useful helper methods to return an instance of **IActionResult**:

- Ok()
- NotFound()
- BadRequest()

Challenge 2

By: Mosh Hamedani

API Inputs

In actions for creating and updating objects, you need to get the JSON object that is sent in the body of the request. Decorate the parameter of these actions with **[FromBody]** attribute:

```
public IActionResult Create([FromBody] Student student)
```

Many-to-many Relationships

In our domain model, we need a many-to-many relationship between vehicles and their features. (In the real-world, it may be more accurate to associate features with models rather than individual vehicles. However, for the purpose of this course, assume that features are dependent on the vehicle.)

At the time of recording this course, Entity Framework Core (v1) does not support many-to-many relationships. But chances are, by the time you're taking this course, this feature is added in the current version of Entity Framework.

So, to implement a many-to-many relationship, we need an association class between vehicles and features. This is exactly the same way we implement many-to-many relationships in relational databases. There's always an intermediary table with a composite primary key (or two foreign keys) referencing the related tables.

In your solution, call this class **VehicleFeature**. This class should have two references: one to the **Vehicle** class and the other to the **Feature** class. Be sure to add primary key properties as well (**VehicleId** and **FeatureId**).

In DbContext, you need to use fluent API to define a composite primary key on **VehicleFeature** class.

```
protected override void OnModelCreating(ModelBuilder mb)
{
   mb.Entity<VehicleFeature>().HasKey(vf =>
        new { vf.VehicleId, vf.FeatureId });
}
```