POST, PUT and DELETE

Handling POST Request

First, let's modify the decoration attribute for the action method `GetOwnerById` in the `Owner` controller:

```
[HttpGet("{id}", Name = "OwnerById")]
```

Setting the name for the action will come in handy in the action method for creating a new owner.

Second, let's modify the interface:

```
public interface IOwnerRepository
{
    IEnumerable<Owner> GetAllOwners();
    Owner GetOwnerById(Guid ownerId);
    OwnerExtended GetOwnerWithDetails(Guid ownerId);
    void CreateOwner(Owner owner);
}
```

After the interface modification, we are going to implement that interface:

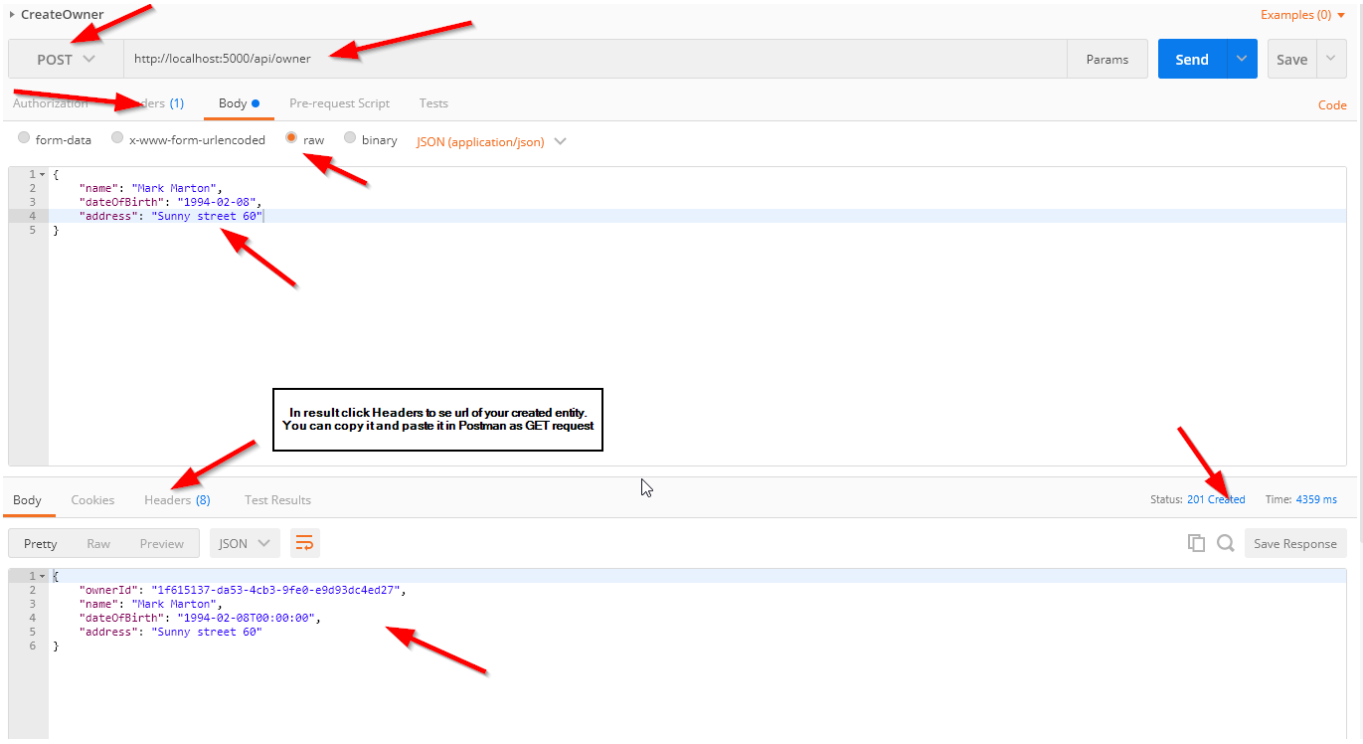
```
public void CreateOwner(Owner owner)
{
    owner.Id = Guid.NewGuid();
    Create(owner);
    Save();
}
```

Finally, let's modify the controller:

```
[HttpPost]
public IActionResult CreateOwner([FromBody]Owner owner)
{
    try
    {
        if(owner == null)
        {
            _logger.LogError("Owner object sent from client is null.");
            return BadRequest("Owner object is null");
        }

        if(!ModelState.IsValid)
        {
```

```
            _logger.LogError("Invalid owner object sent from client.");
            return BadRequest("Invalid model object");
        }

        _repository.Owner.CreateOwner(owner);

        return CreatedAtRoute("OwnerById", new { id = owner.Id}, owner);
    }
    catch (Exception ex)
     {
        _logger.LogError($"Something went wrong inside CreateOwner action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

Test the code by sending a POST request using Postman.



Code Explanation

The code in the controller contains several things worth mentioning.

The `CreateOwner` method has an `[HttpPost]` decoration attribute, which restricts it to POST requests. Furthermore, notice the `owner` parameter that comes from the client. You are not collecting it from the Uri but from the request body. Thus the usage of the `[FromBody]` attribute. Also, the owner object is a complex type and because of that, you have to use `[FromBody]`.

Because the `owner` parameter comes from the client, it could happen that the client doesn't send that parameter at all. As a result, you need to validate it against the reference type's default value, which is `null`. Later, we are going to refactor this line of code.

Further down, you see this validation: `if (!ModelState.IsValid)`. If you look at the owner model properties: `Name`, `Address`, and `DateOfBirth`, you see that all of them are decorated with validation attributes. If validation fails, `ModelState.IsValid` will return false, signaling that something is wrong with the model. Otherwise, it will return true which means that values in all the properties are valid.

The last thing to mention is this code:

```
CreatedAtRoute("OwnerById", new { id = owner.Id }, owner);
```

`CreatedAtRoute` will return a status code 201, which stands for `Created`. Also, it will populate the body of the response with the new owner object as well as the `Location` attribute within the response header with the address to retrieve that owner. You need to provide the name of the action, where you can retrieve the created entity.

Code Refactoring

Let's do some refactoring to help with the `null` validation of the `CreateOwner` method. It will also help with the validation of the `GET` actions.

Create a new interface in the `Entities` project:

```
namespace Entities
{
    public interface IEntity
    {
        Guid Id { get; set; }
    }
}
```

Then modify the model classes -- `Owner`, `OwnerExtended` and `Account` -- to implement that interface. In the same project, we are going to create the folder `Extensions` and in that folder create the new class `EntityExtensions`:

```
namespace Entities.Extensions
{
    public static class EntityExtensions
    {

    }
}
```

Add an extension method for the `IEntity` type to validate against `null` value. Because `Owner` and the `Account` inherit from `IEntity`, we can extend any of those types and validate them.

```
public static class EntityExtensions
{
    public static bool IsObjectNull(this IEntity entity)
    {
        return entity == null;
    }
}
```

Now in the `CreateOwner` action method, replace this code:

```
if (owner == null)
```

with this:

```
if (owner.IsObjectNull())
```

Let's refactor this code a little bit more.

Create another extension method:

```
public static bool IsEmptyObject(this IEntity entity)
{
    return entity.Id.Equals(Guid.Empty);
}
```

Now change the code in the controller from the:

```
if (owner.OwnerId.Equals(Guid.Empty))
```

to this:

```
if (owner.IsEmptyObject())
```

in the `GetOwnerWithDetails` and `GetOwnerById` actions.

Handling PUT Request

Let's continue with the `PUT` request to update the owner entity.

First, in the `Entities` project, in the `Extensions` folder, we are going to create the `OwnerExtensions` class:

```
namespace Entities.Extensions
{
    public static class OwnerExtensions
    {
        public static void Map(this Owner dbOwner, Owner owner)
        {
            dbOwner.Name = owner.Name;
            dbOwner.Address = owner.Address;
            dbOwner.DateOfBirth = owner.DateOfBirth;
        }
    }
}
```

Then change the interface:

```
public interface IOwnerRepository
{
    IEnumerable<Owner> GetAllOwners();
    Owner GetOwnerById(Guid ownerId);
    OwnerExtended GetOwnerWithDetails(Guid ownerId);
    void CreateOwner(Owner owner);
    void UpdateOwner(Owner dbOwner, Owner owner);
}
```

Modify `OwnerRepository.cs`:

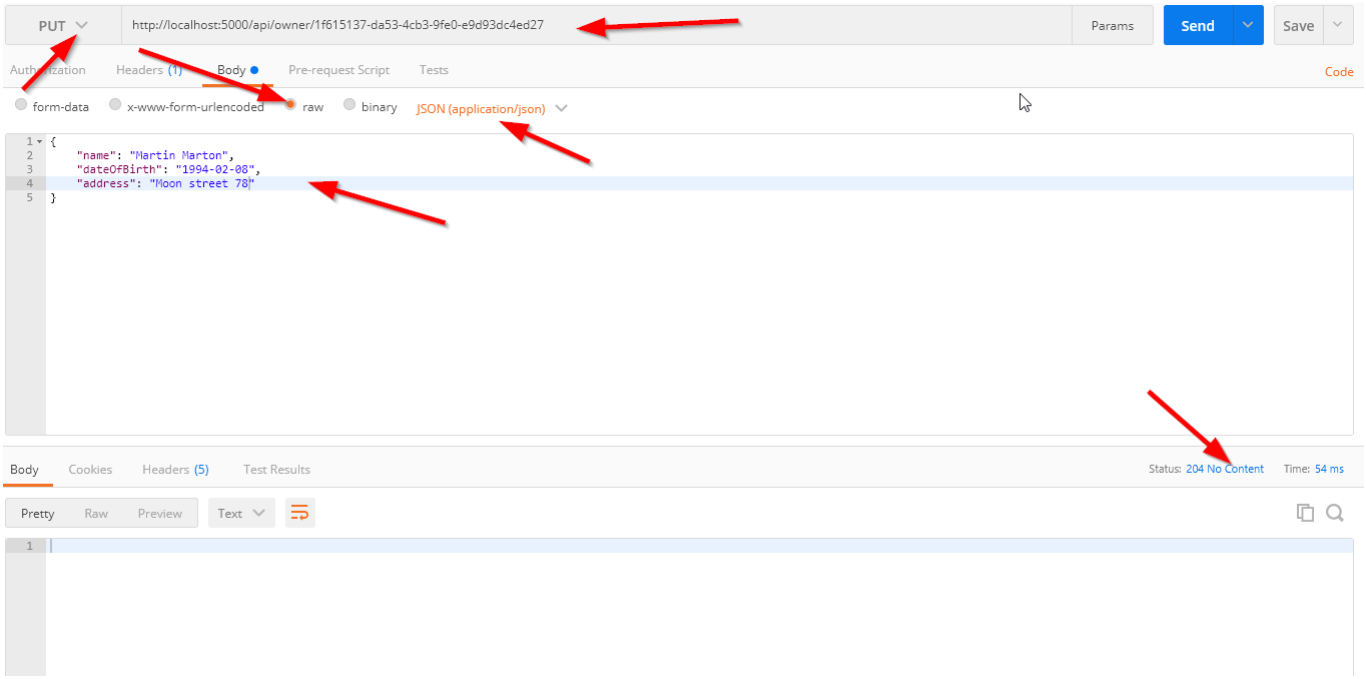
```
public void UpdateOwner(Owner dbOwner, Owner owner)
{
    dbOwner.Map(owner);
    Update(dbOwner);
    Save();
}
```

Finally, update the `OwnerController`:

```
[HttpPut("{id}")]
public IActionResult UpdateOwner(Guid id, [FromBody]Owner owner)
{
    try
    {
        if (owner.IsObjectNull ())
        {
            _logger.LogError("Owner object sent from client is null.");
            return BadRequest("Owner object is null");
        }
```

```
            if (!ModelState.IsValid)
            {
                _logger.LogError("Invalid owner object sent from client.");
                return BadRequest("Invalid model object");
            }

            var dbOwner = _repository.Owner.GetOwnerById(id);
            if (dbOwner.IsEmptyObject())
            {
                _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
                return NotFound();
            }

            _repository.Owner.UpdateOwner(dbOwner, owner);

            return NoContent();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong inside UpdateOwner action:
    {ex.Message}");
            return StatusCode(500, "Internal server error");
        }
    }
```

The action method is decorated with the `[HttpPut]` attribute. It receives two parameters: `id` of the entity you want to update and the entity with the updated fields, taken from the request body. After validation, you fetch the owner from the database and update that owner. Finally, return `NoContent` which is a status code 204.



Handling DELETE Request

Modify the interface:

```
public interface IOwnerRepository
{
    IEnumerable<Owner> GetAllOwners();
    Owner GetOwnerById(Guid ownerId);
    OwnerExtended GetOwnerWithDetails(Guid ownerId);
    void CreateOwner(Owner owner);
    void UpdateOwner(Owner dbOwner, Owner owner);
    void DeleteOwner(Owner owner);
}
```

Modify `OwnerRepository`:

```
public void DeleteOwner(Owner owner)
{
    Delete(owner);
    Save();
}
```

Modify `OwnerController`:

```
[HttpDelete("{id}")]
public IActionResult DeleteOwner(Guid id)
{
    try
    {
        var owner = _repository.Owner.GetOwnerById(id);
        if(owner.IsEmptyObject())
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
            return NotFound();
        }

        _repository.Owner.DeleteOwner(owner);

        return NoContent();
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside DeleteOwner action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

Let's handle one more thing. If we try to delete an owner that has accounts, we are going to get an error because we didn't allow cascade delete in our database configuration. What we want is to return a `BadRequest` error, which requires some modifications.

Modify the `IAccountRepository` interface:

```
using Entities.Models;
using System;
using System.Collections.Generic;

namespace Contracts
{
    public interface IAccountRepository
    {
        IEnumerable<Account> AccountsByOwner(Guid ownerId);
    }
}
```

Then modify the `AccountRepository` by adding one new method:

```
public IEnumerable<Account> AccountsByOwner(Guid ownerId)
{
    return FindByCondition(a => a.OwnerId.Equals(ownerId));
}
```

Finally, modify the `DeleteOwner` action in the OwnerController by adding one more validation before deleting owner:

```
if(_repository.Account.AccountsByOwner(id).Any())
{
    _logger.LogError($"Cannot delete owner with id: {id}. It has related accounts.
Delete those accounts first");
    return BadRequest("Cannot delete owner. It has related accounts. Delete those
accounts first");

}
```

Send the `DELETE` request from Postman and see the result. Owner object should be deleted from the database.

Conclusion

Now that you know all of this, try to repeat all the actions but for the `Account` entity.

With all this code in place, we have a working Web API that covers all the features for handling CRUD operations.

Lessons learned:

- How to handle POST requests
- How to handle PUT requests

- How to write better and more reusable code
- How to handle DELETE requests