9 CREATING RESOURCES

In this section, we are going to show you how to use the POST HTTP method to create resources in the database.

So, let's start.

9.1 Handling POST Requests

Firstly, let's modify the decoration attribute for the `GetCompany` action in the `Companies` controller:

```
[HttpGet("{id:guid}", Name = "CompanyById")]
```

With this modification, we are setting the name for the action. This name will come in handy in the action method for creating a new company.

We have a DTO class for the output (the GET methods), but right now we need the one for the input as well. So, let's create a new record in the `Shared/DataTransferObjects` folder:

```
public record CompanyForCreationDto(string Name, string Address, string Country);
```

We can see that this DTO record is almost the same as the `Company` record but without the Id property. We don't need that property when we create an entity.

We should pay attention to one more thing. In some projects, the input and output DTO classes are the same, but we still recommend separating them for easier maintenance and refactoring of our code. Furthermore, when we start talking about validation, we don't want to validate the output objects — but we definitely want to validate the input ones.

With all of that said and done, let's continue by modifying the `ICompanyRepository` interface:

```
public interface ICompanyRepository
{
    IEnumerable<Company> GetAllCompanies(bool trackChanges);
```

```
    Company GetCompany(Guid companyId, bool trackChanges);
    void CreateCompany(Company company);
}
```

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

```
public void CreateCompany(Company company) => Create(company);
```

We don't explicitly generate a new Id for our company; this would be done by EF Core. All we do is to set the state of the company to Added.

Next, we want to modify the **ICompanyService** interface:

```
public interface ICompanyService
{
       IEnumerable<CompanyDto> GetAllCompanies(bool trackChanges);
       CompanyDto GetCompany(Guid companyId, bool trackChanges);
       CompanyDto CreateCompany(CompanyForCreationDto company);
}
```

And of course, we have to implement this method in the **CompanyService** class:

```
public CompanyDto CreateCompany(CompanyForCreationDto company)
{
       var companyEntity = _mapper.Map<Company>(company);

       _repository.Company.CreateCompany(companyEntity);
       _repository.Save();

       var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

       return companyToReturn;
}
```

Here, we map the company for creation to the company entity, call the repository method for creation, and call the **Save()** method to save the entity to the database. After that, we map the company entity to the company DTO object to return it to the controller.

But we don't have the mapping rule for this so we have to create another mapping rule for the **Company** and **CompanyForCreationDto** objects. Let's do this in the **MappingProfile** class:

```
public MappingProfile()
{
    CreateMap<Company, CompanyDto>()
      .ForMember(c => c.FullAddress,
```

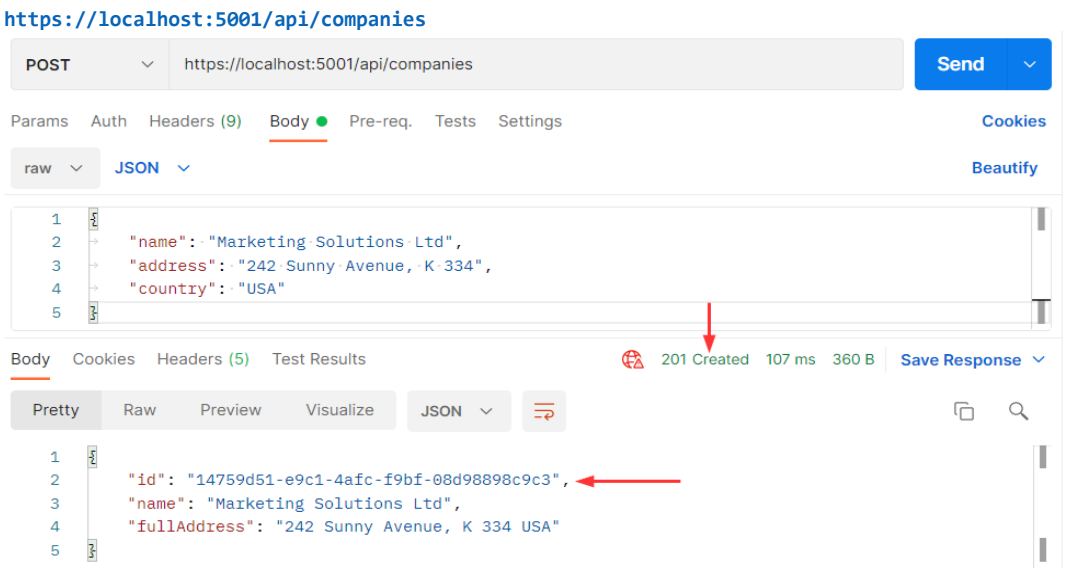
```
            opt => opt.MapFrom(x => string.Join(' ', x.Address, x.Country)));

        CreateMap<Employee, EmployeeDto>();

        CreateMap<CompanyForCreationDto, Company>();
    }
```

Our POST action will accept a parameter of the type `CompanyForCreationDto`, and as you can see our service method accepts the parameter of the same type as well, but we need the `Company` object to send it to the repository layer for creation. Therefore, we have to create this mapping rule.

Last, let's modify the controller:

```
[HttpPost]
public IActionResult CreateCompany([FromBody] CompanyForCreationDto company)
{
       if (company is null)
              return BadRequest("CompanyForCreationDto object is null");

       var createdCompany = _service.CompanyService.CreateCompany(company);

       return CreatedAtRoute("CompanyById", new { id = createdCompany.Id },
createdCompany);
}
```

Let's use Postman to send the request and examine the result:



9.2 Code Explanation

Let's talk a little bit about this code. The interface and the repository parts are pretty clear, so we won't talk about that. We have already explained the code in the service method. But the code in the controller contains several things worth mentioning.

If you take a look at the request URI, you'll see that we use the same one as for the GetCcompanies action: `api/companies` — but this time we are using the POST request.

The `CreateCompany` method has its own `[HttpPost]` decoration attribute, which restricts it to POST requests. Furthermore, notice the company parameter which comes from the client. We are not collecting it from the URI but the request body. Thus the usage of the `[FromBody]` attribute. Also, the company object is a complex type; therefore, we have to use `[FromBody].`

If we wanted to, we could explicitly mark the action to take this parameter from the URI by decorating it with the `[FromUri]` attribute, though we wouldn't recommend that at all because of security reasons and the complexity of the request.

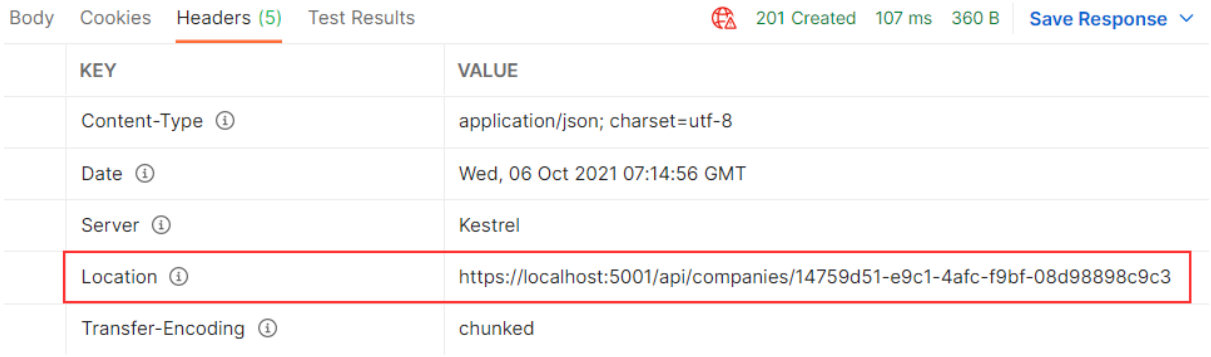
Because the `company` parameter comes from the client, it could happen that it can't be deserialized. As a result, we have to validate it against the reference type's default value, which is null.

The last thing to mention is this part of the code:

```
CreatedAtRoute("CompanyById", new { id = companyToReturn.Id }, companyToReturn);
```

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

If we take a look at the headers part of our response, we are going to see a link to retrieve the created company:



Finally, from the previous example, we can confirm that the POST method is neither safe nor idempotent. We saw that when we send the POST request, it is going to create a new resource in the database — thus changing the resource representation. Furthermore, if we try to send this request a couple of times, we will get a new object for every request (it will have a different Id for sure).

Excellent.

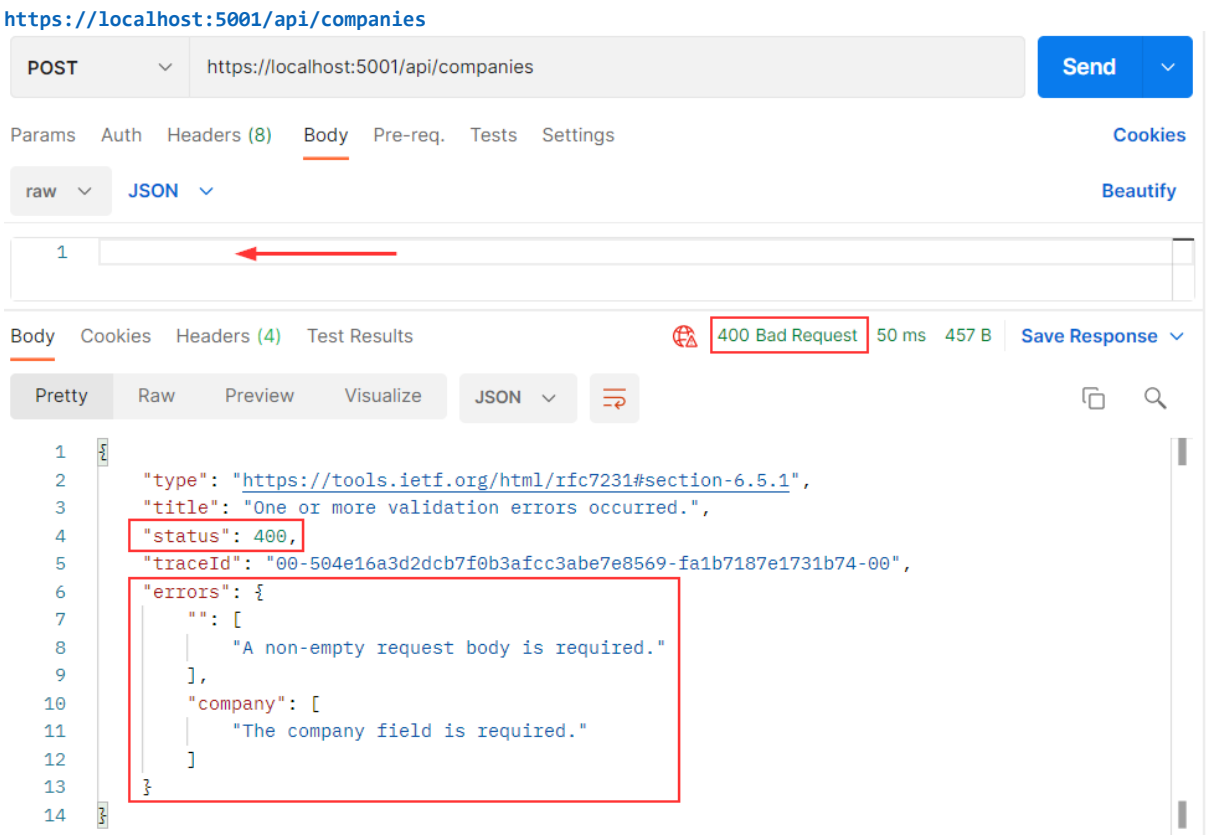
There is still one more thing we need to explain.

9.2.1 Validation from the ApiController Attribute

In this section, we are going to talk about the [ApiController] attribute that we can find right below the [Route] attribute in our controller:

```
[Route("api/companies")]
[ApiController]
public class CompaniesController : ControllerBase
{
```

But, before we start with the explanation, let's place a breakpoint in the `CreateCompany` action, right on the `if (company is null)` check. Then, let's use Postman to send an invalid POST request:



We are going to talk about Validation in chapter 13, but for now, we have to explain a couple of things.

First of all, we have our response - a Bad Request in Postman, and we have error messages that state what's wrong with our request. But, we never hit that breakpoint that we've placed inside the `CreateCompany` action.

Why is that?

Well, the `[ApiController]` attribute is applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference
- Problem details for error status codes

As you can see, it handles the HTTP 400 responses, and in our case, since the request's body is null, the `[ApiController]` attribute handles that and returns the 400 (BadReqeust) response before the request even hits the `CreateCompany` action.

This is useful behavior, but it prevents us from sending our custom responses with different messages and status codes to the client. This will be very important once we get to the Validation.

So to enable our custom responses from the actions, we are going to add this code into the `Program` class right above the `AddControllers` method:

```
builder.Services.Configure<ApiBehaviorOptions>(options =>
{
      options.SuppressModelStateInvalidFilter = true;
});
```

With this, we are suppressing a default model state validation that is implemented due to the existence of the `[ApiController]` attribute in all API controllers. So this means that we can solve the same problem differently, by commenting out or removing the `[ApiController]` attribute only, without additional code for suppressing validation. It's all up to you. But we like keeping it in our controllers because, as you could've seen, it provides additional functionalities other than just 400 – Bad Request responses.

Now, once we start the app and send the same request, we will hit that breakpoint and see our response in Postman.

Nicely done.

Now, we can remove that breakpoint and continue with learning about the creation of child resources.

9.3 Creating a Child Resource

While creating our company, we created the DTO object required for the CreateCompany action. So, for employee creation, we are going to do the same thing:

```
public record EmployeeForCreationDto(string Name, int Age, string Position);
```

We don't have the **Id** property because we are going to create that Id on the server-side. But additionally, we don't have the **CompanyId** because we accept that parameter through the route:

```
[Route("api/companies/{companyId}/employees")]
```

The next step is to modify the **IEmployeeRepository** interface:

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);
    Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);
    void CreateEmployeeForCompany(Guid companyId, Employee employee);
}
```

Of course, we have to implement this interface:

```
public void CreateEmployeeForCompany(Guid companyId, Employee employee)
{
    employee.CompanyId = companyId;
    Create(employee);
}
```

Because we are going to accept the employee DTO object in our action and send it to a service method, but we also have to send an employee object to this repository method, we have to create an additional mapping rule in the **MappingProfile** class:

```
CreateMap<EmployeeForCreationDto, Employee>();
```

The next thing we have to do is **IEmployeeService** modification:

```
public interface IEmployeeService
{
       IEnumerable<EmployeeDto> GetEmployees(Guid companyId, bool trackChanges);
       EmployeeDto GetEmployee(Guid companyId, Guid id, bool trackChanges);
       EmployeeDto CreateEmployeeForCompany(Guid companyId, EmployeeForCreationDto
employeeForCreation, bool trackChanges);
}
```

And implement this new method in EmployeeService:

```
public EmployeeDto CreateEmployeeForCompany(Guid companyId, EmployeeForCreationDto
employeeForCreation, bool trackChanges)
{
	var company = _repository.Company.GetCompany(companyId, trackChanges);
	if (company is null)
		throw new CompanyNotFoundException(companyId);

	var employeeEntity = _mapper.Map<Employee>(employeeForCreation);

	_repository.Employee.CreateEmployeeForCompany(companyId, employeeEntity);
	_repository.Save();

	var employeeToReturn = _mapper.Map<EmployeeDto>(employeeEntity);

	return employeeToReturn;
}
```

We have to check whether that company exists in the database because there is no point in creating an employee for a company that does not exist. After that, we map the DTO to an entity, call the repository methods to create a new employee, map back the entity to the DTO, and return it to the caller.

Now, we can add a new action in the **EmployeesController**:

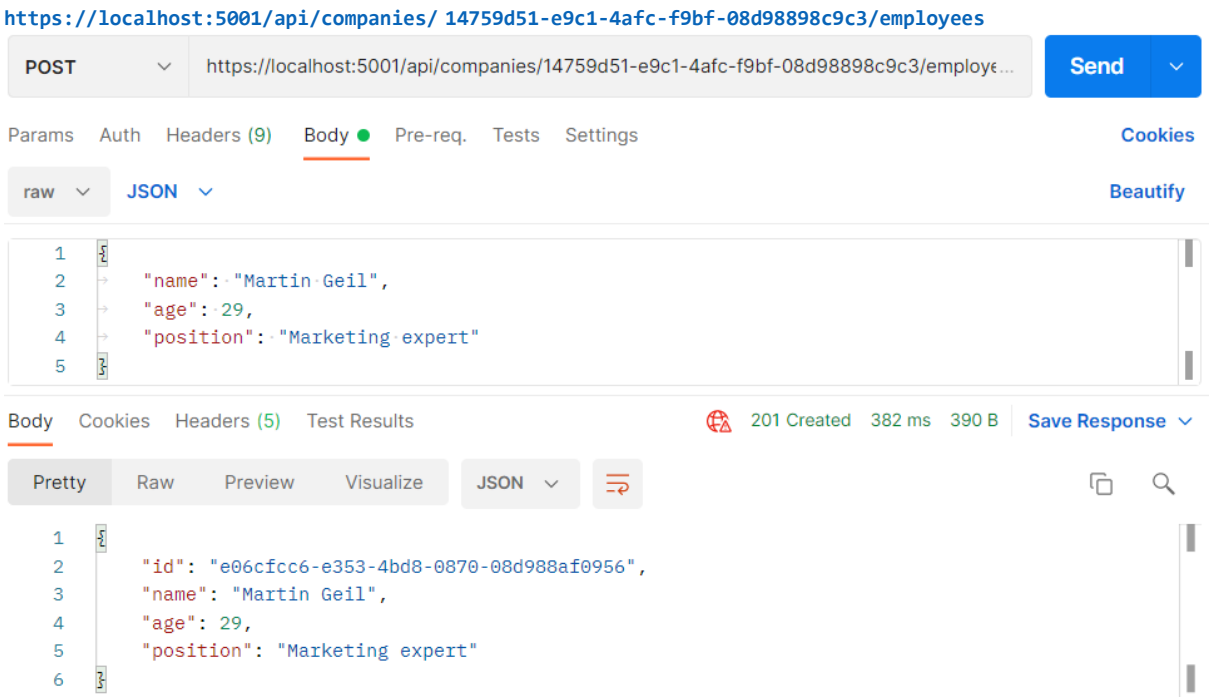
```
[HttpPost]
public IActionResult CreateEmployeeForCompany(Guid companyId, [FromBody]
EmployeeForCreationDto employee)
{
	if (employee is null)
		return BadRequest("EmployeeForCreationDto object is null");

	var employeeToReturn =
_service.EmployeeService.CreateEmployeeForCompany(companyId, employee, trackChanges:
false);

	return CreatedAtRoute("GetEmployeeForCompany", new { companyId, id =
employeeToReturn.Id },
		employeeToReturn);
}
```

As we can see, the main difference between this action and the **CreateCompany** action (if we exclude the fact that we are working with different DTOs) is the return statement, which now has two parameters for the anonymous object.

For this to work, we have to modify the HTTP attribute above the `GetEmployeeForCompany` action:

```
[HttpGet("{id:guid}", Name = "GetEmployeeForCompany")]
```

Let's give this a try:



Excellent. A new employee was created.

If we take a look at the Headers tab, we'll see a link to fetch our newly created employee. If you copy that link and send another request with it, you will get this employee for sure: