What I did

Refactor

Onion architecture

I implemented a DDD-friendly architecture (so called Onion – I can explain in more details).
Basically:
- SynetecAssessmentApi.Domain
This becomes the Core layer – it's the layer where the Model and Interfaces are defined. This is the place where business services are defined – but these services are agnostic about the infrastructure (data persistence, email sending,…). See BonusPoolService and EmployeeService: these guys know, for example, that they have to return an Employee based on its ID by calling a repository (EmployeeService).

- SynetecAssessment.Persistence
I'm not happy with the name. I would call it *Infrastructure* – and this is the place where the Interfaces dealing with database (but not only) are implemented. For example, a generic repository is defined in Core, but it's implemented in this project.

- SynetecAssessmentApi
It's the project containing the controllers.

Interfaces

I made all the services (and the DbContext) injectable.

Refactored BonusPoolService

This guy was doing too many things: get all employees from database (??), get one employee from database, calculate the bonus, returning a DTO… I kept only the calculation here.
! This has only one method accepting various decimals (btw, I prefered decimals instead of integers because, when working with money, it's safer to avoid rounding the numbers as late as possible). Now, when I am thinking about it, I think a better approach would have been to send an Employee as input, and not just some numbers, because the idea is to calculate the bonus for en Employee and not to perform just some arithmetical operations. To be discussed.

Unit tests

Added Unit tests for:
- BonusPoolService – I tried to cover also the edge cases, like, for example, negative numbers or division by zero. To enforce these rules I added a nuget package Ardalis.GuardClause and I'm checking it in the BonusPoolService.

- EmployeeService

Nothing fancy here, an improvement can be reusing the creation of test data (see GetTestEmployee method from BonusPoolController)

- BonusPoolController
Here I changed a little bit the requirements. If the employee doesn't exist, I'm returning a NoContent instead of BadRequest.
I added an additional test which verifies that BonusPoolService.CalculateBonusAsync is indeed called

Other possible improvements:
- use FluentAssertions
- introduce integration tests for controller (not required :p)

Other stuff

BonusPoolController

Right now the CalculateBonusAsync is using two dependencies to calculate the bonus: EmployeeService is returning the Employee, BonusPoolService is calculating the bonus.
Another approach could have been that the EmployeeService is just receiving the ID of the Employee, calls internally the BonusPoolService in order to find the bonus amount and then return *something* that can be used to send data to client. I didn't like this approach because this means that I have to "pollute" the domain – I have to return a something (a mutant between the Employee and the amount value) which has no meaning in the domain. One way to do it is to use out parameters (ugly!), a tuple (not so ugly, but still). Another way would have been to adjust the Employee to accommodate the Amount (not convinced at all that the amount should be part of the domain).
Also I introduced here the FluentValidation nuget package (see Validators/CalculateBonusDtoValidator) which helps on enforcing the model validation. Currently I am returning a list of errors by flattening whatever ModelState gives me back.
Also I am using here a static mapper (using a nice Visual Studio extension). Another way can be to use AutoMapper – but I think it would have been to much trouble.

AppDbContext

I didn't touch it, I just moved it under SynetecAssessment.Persistence/Data. One note here: I'm not a big fan of InMemoryDatabase (I think it has a limitation when it comes to cascade deleting entities – but I'm not sure).

EFRepository<T>

I kept this slim because of YAGNI. Usually in a repository there should be more functionality (for adding, updating, deleting, plus a way to execute specific queries over the database). It was not the case for this app.

EmployeeRepository

Inherits from EFRepository, but I am overriding the GetAllSync and GetByIdAsync because I need to include the Department and also I need an extra thing there, namely GetTotalSalaries.

DbContextGenerator

I made the SeedData private so that I'll prevent potential callers to seed data again. In fact, it was not necessary at all to be public. A note about this: I prefer migrations for this data seeding.

Startup

I added the services for DI and removed UseAuthorization (since it was not required).

IRepository<T>

As I already wrote above, I kept it to a minimum (YAGNI).

All

I updated to .NET Core 6 – not required, but it's nice to use the newest tools. I hope you can still run it.