Alright, so here in the Mocking folder let's add a new class EmployeeStorage. Some people may prefer to call this EmployeeService, but I personally prefer not to use the word service, because service is over loaded in the community, it can mean different things, it also has different meanings depending on what layer it's used in, I personally use services to implement an application service layer.

An application service layer is a layer around your application that is responsible for the high level orchestration. So it delegates tasks to different objects. For example, it may talk to an object to store something in a database, and then it may talk to another object to send a notification to the user, and then finally it may talk to a logger object to write something to the log.

So this is what we call high level orchestration. And when I have scenarios like that, I use the word service. But in this case we don't have high level orchestration, we don't need an application service layer all you're doing is deleting an employee from our database. That's why I prefer to call this class EmployeeStorage.

Because it's purely responsible for storage. Now you might ask why I didn't call this EmployeeRepository. Well, we could certainly do that, but repositories should not have a save method. So we can not call respository.save changes. I've explained the reason for that in my editing framework course I've also written block posts about that. So simply search for Mosh Hamedani, repository pattern. You will see I have a few blog posts about repository and how you should use them. So if you want to use a repository pattern here, you should also bring in a unit award (?) and that makes this design a little bit more complicated.

So that's why I prefer to use an EmployeeStorage instead of EmployeeRepository and unit award (?) in this context. So, EmployeeStorage should have one method, DeleteEmployee, it takes an integer. Now, back to our controller, I'm going to take these three lines. Cut.

And move it inside this method. So here you can see we need a reference to our db context, so I'm going to go back to our controller and move this private field into our storage class as well.

And finally, let's initialize this in the constructor. So a new EmployeeContext. And by the way, this code has a bug, because on this line it is possible that we don't find an employee with this ID. So when we pass null here, we're going to get an exception. So I'm going to modify this to if employee is not null, then we're going to remove it from context, and save the changes, okay? Or, another way to rewrite this code is to reverse this conditional statement and then you can get rid of this code block and this extra indentation. Let me show you what I mean. So, if employee is null, then we can return immediately, right? And with that we no longer need this code block, and this extra indentation.

Like this. Okay, so this is our employee storage, it's purely responsible for working with our database. Now today in this implementation we're using any (?) framework to achieve that. Maybe tomorrow we'll decide to use a different framework, or you might decide to use plain sequel connection and sequel commands to delete an employee from a database. That's implementation detail and can change in the future. Our controller doesn't care about this implementation detail, as long as this EmployeeStorage has a mechanism to delete an employee, our controller is going to be happy. So, now we need to extract an interface from this class. So alt and enter. Refactor

Extract interface. And Let's add this member. Okay, done. So, here's our new interface, now we need to inject this into the constructor of our EmployeeController, so, back here, here in the constructor IEmployeeStorage. Let's call that storage, and create and initialize a private field in one go.

So we no longer need to new up this employee. Context instead we have a private field of type IEmployeeStorage. Now finally, in our DeleteEmployee method, we need to call storage.DeleteEmployee and pass this ID here. So we can see with this refactoring we have proper separation of concerns. Nowhere in our controller we are touching our database, all that responsibility is encapsulated inside our storage.