Alright, so back in our solution explorer, inside the Mocking folder, let's add a new class called BookingRepository. So we're going to encapsulate that query inside a repository class. Or you can call this BookingStorage. Whatever you prefer.

So, here we should have a method, that returns an IQueryable object. of booking. Why IQueryable? because if you look at the booking helper class, look at this implementation. It's using unitOfWork.Query

of Booking, and here's the logic for the query.

So this method returns an IQueryableObject, and further below, we are customizing that query even further. Okay? So, back in Booking Repository, we should return an IQueryable of Booking and let's call this method GetActiveBookings. Now, back in the BookingHelper class, let's move these few lines where we touch the database, inside our BookingRepository.

And here we return bookings. Now here we have a compilation issue, and we're going to fix that in a second, but I want to tell you another way to refactor this code. Instead of extracting these few lines, we could extract an interface from unit of work class, so instead of instantiating this unit of work inside our BookingHelper here, we could inject a IUnit of work, then, in our unit tests, we could mock the Unit of work interface.

So that approach is perfectly fine. However, the reason I extracted these few lines, is because I want to encapsulate this query inside this method. And this is for better separation of concerns. Because it is possible that somewhere else in the application, you want to get the list of active bookings. Perhaps you want to display them somewhere for admins. In that case, we don't want to repeat this logic specifically the part where we check the status of the booking. We don't want to repeat this in multiple places. And that's why we use the repository pattern to encapsulate queries and put them where where they really belong. Now you might ask, how come

I didn't extract these additional lines here, because this is the logic specifically for this method. Because once we put active bookings, then we can see if there is an overlap between any of those bookings and the booking object we pass here. Okay? So, back in our booking repository, here we have this compilation issue, because we don't have the ID of this booking. So, I'm going to add a parameter to this method, we only need an ID, not a booking object, so I want to pass, an integer, and call that excluded booking ID.

Now, it is possible, that somewhere else in our application, we want to call this method get active bookings without passing a booking ID to exclude. We just want to see all the active bookings in the application. So, I want to make this nullible and set it to null, and this way this will be an optional argument. Now, we need to modify this query like this. So, first we check only for the status of the booking, let's delete the first part, where we check the ID of the booking, now if our parameter has a value, so excludedBookingiD.HasValue will further customize this query.

So, we set bookings, which is an IQueryable object, to bookings.Where, and here we write some additional querying logic. So b goes to, and now we paste the part that we cut. And finally, let's modify this. You want to compare, b.Id with excluded booking ID.value.

So with this technique, I can construct dynamic queries. And by the way, here I actually cheated, because I shouldn't have modified the logic in this method, without covering it with an integration test. Because it is possible that you make a mistake as part of breaking this logic in to some kind of dynamically constructed logic.

However, because our focus is on unit tests in this course, and integration tests is a topic for another course, we cannot write an integration test for this method. But once you learn integration testing, in your applications, before you do any refactoring, it's always best to write enough to write enough to write proper unit and or integration tests for a method, to make sure, as part of your refactoring you're not breaking anything. Okay.

So, here we have a repository class that encapsulates the logic for getting active queries. Now we need to extract an interface from this class. So alt and enter. Refactor, and extract interface. Let's add this method here, next, and here's our interface. Beautiful.

Now, back in our Booking Helper class, we need to inject an instance of a class that implements IBookingRepository interface. As you learned before, you can inject it as a parameter here, or using a property, or using a constructor for this class. However, you can see that this method is declared as static.

So we can't use constructor, or property injection here. So we have two options. One is to inject this dependency using a parameter in this method, or remove the static modifier, and make this method an instance method, and then we can use constructor injection. Most of these approaches heave problems. The problem with the first approach is that the dependency injection framework you use may not support parameter injection. It may only support constructor injection. If that's the case, you really need to modify this method and remove the static modifier.

But the problem with this approach is that maybe you have used this method in a few different places in the application, and you have to go and modify all those places. And chances are you may break something along the way.

So these are our constraints, for the rest of the section I'm assuming that the dependency injection framework you potentially use in our application does support parameter injection. And it is too costly to modify this method, and make it an instance method. So, with that assumption, I'm going to add IBookingRepository as a parameter here, and call it repository.

Then, we can get all the active bookings. So var bookings, set it to repository.GetActiveBookings and we pass booking.Id as an argument here. So now, our method is loosely coupled, and it's ready for testing. In the next lecture, we're going to write our first unit test.