Edit

Creating Unit Tests for Person Application Service

You can skip this section if you are not interested in **automated testing**.

By writing unit test, we can test **PersonAppService.GetPeople** method without creating a user interface. We write unit test in .**Tests** project in the solution.

MultiTenancy In Tests

Since we disabled multitenancy, we should disable it for unit tests too. Open **PhoneBookDemoConsts** class in the Acme.PhoneBook.Core project and set "**MultiTenancyEnabled**" to false. After a rebuild and run unit tests, you will see that some tests are skipped (those are related to multitenancy).

Let's create first test to verify getting people without any filter:

C# Copy

```
using Acme.PhoneBookDemo.People;
using Acme.PhoneBookDemo.People.Dtos;
using Shouldly;
using Xunit;

namespace Acme.PhoneBookDemo.Tests.People
{
    public class PersonAppService_Tests : AppTestBase
    {
        private readonly IPersonAppService _personAppService;

        public PersonAppService_Tests()
        {
            _personAppService = Resolve<IPersonAppService>();
        }

        [Fact]
        public void Should_Get_All_People_Without_Any_Filter()
```

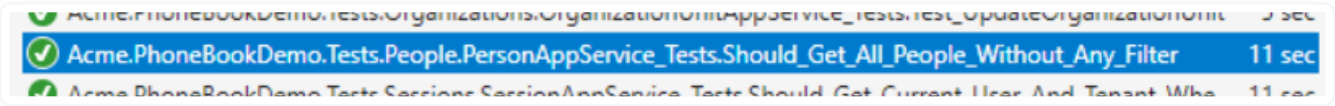
```
            //Assert
            persons.Items.Count.ShouldBe(2);
        }
    }
}
```

We derived test class from **AppTestBase**. AppTestBase class initializes all system, creates an in-memory fake database, seeds initial data (that we created before) to database and logins to application as admin. So, this is actually an **integration test** since it tests all server-side code from entity framework mapping to application services, validation and authorization.

In constructor, we get (resolve) an **IPersonAppService** from **dependency injection** container. It creates the **PersonAppService** class with all dependencies. Then we can use it in test methods.

Since we're using xUnit, we add **Fact** attribute to each test method. In the test method, we called **GetPeople** method and checked if there are **two people** in the returned list as we know that there were 2 people in **initial** database.

Let's run the **all unit tests** in Test Explorer and see if it works:



As you see, it worked **successfully**. Now, we know that PersonAppService works properly without any filter. Let's add a new unit test to get filtered people:

```
[Fact]
public void Should_Get_People_With_Filter()
{
    //Act
    var persons = _personAppService.GetPeople(
        new GetPeopleInput
        {
            Filter = "adams"
        });

    //Assert
    persons.Items.Count.ShouldBe(1);
    persons.Items[0].Name.ShouldBe("Douglas");
```

Again, since we know initial database, we can check returned results easily. Here, initial test data is important. When we change initial data, our test may fail even if our services are correct. So, it's better to write unit tests independent of initial data as much as possible. We could check incoming data to see if every people contains "adams" in his/her name, surname or email. Thus, if we add new people to initial data, our tests remain working.

There are many techniques on unit testing, I kept it simple here. But ASP.NET Zero template makes very easy to write unit and integration tests by base classes and pre-build test codes.

Next

- Creating Person Application Service