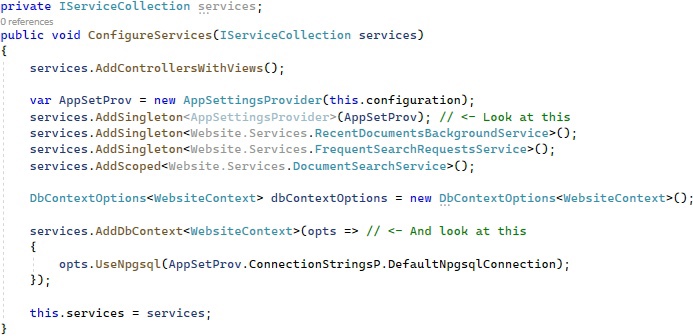
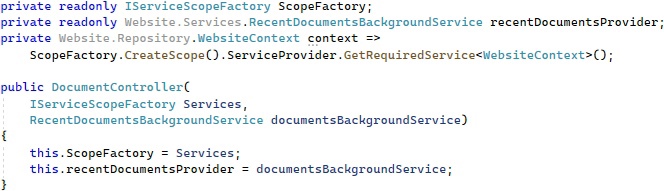
**Микросервисная архитектура в MVC.NET**Внедрение зависимостей и инверсия управления.

В ходе разработки своего первого относительно большого проекта на ASP я впервые столкнулся с необходимостью работать с большим количеством классов, имеющих разный lifetime, которые являются разными частями приложения и отвечают за разные вещи. Естественно, полностью удерживать образ такой системы в голове и не таскать все классы по всему приложению может быть непросто. Именно поэтому в ASP присутствует такой великолепный принцип ООП, как инверсия управления. Она реализуется посредствам, в том числе, внедрения зависимостей. Что такое инверсия управления? — я бы использовал в качестве определения достаточно популярную фразу — «Инверсия управления — это когда не программист управляет фреймворком, а фреймворк управляет кодом программиста». Проще говоря, вы пишите код, а фреймворк определяет, как его встроить в систему. Если более конкретно — вы пишите код контроллера, но нигде его не вызываете. Он вызывается фреймворком автоматически, когда на сайт приходит соответствующий данному контроллеру запрос. Но это есть инверсия управления. Где появляется внедрение зависимостей? В принципе, можно привести тот же контроллер. Да, теоретически, без внедрения зависимостей здесь можно обойтись, но в чем тогда смысл программировать на ASP? Зачем отказываться от легчайшего способа получать экземпляры тех или иных классов, в конструктор которых уже переданы нужные данные? К делу. Начнем с класса Startup и метода ConfigureServices:



Создать карусель

На самом деле в нем мы уже имеем внедренную зависимость, но её пока опустим. Итак, по комментариям вы можете понять, куда следует обратить внимание. В первом «Look at this» я лишь добавляю в приложение службу по извлечению параметров из файла appsettings.json. Но строчкой выше вы видите, что добавляемую службу я присвоил переменной. Всё дело в том, что до выхода из метода, все эти «службы» — не более чем ссылки (кроме AppSettingProvider, это уже экземпляр) на классы, добавленные в IServiceCollection , которая, как понятно из названия, является наследником ICollection. В общем, просто добавляем классы в коллекцию. Но далее мы видим, что переменная AppSetProv используется с целью передачи её поля в метод AddDbContext. Но на самом деле! — мы передаем это поле вовсе не в этот метод. Мы передаём его в конструктор класса DbContext. В каждый из его конструкторов. В этом моменте мы говорим ASP, что ему следует передавать в конструктор при создании каждого экземпляра DbContext. Теперь перейдем в контроллер:



В конструктор контроллера мы передаем IServiceCollection. Помните? Именно ту коллекцию, в которую мы добавляли сервисы в классе Startup. Можно передавать и отдельные службы, как например documentsBackgroundService. В чем состоит внедрение зависимостей? — Нигде в коде мы не создаем экземпляр контроллера. Фреймворк сделает это сам. И сам передаст в него нужные параметры. Здесь контроллер начинает управлять созданием экземпляров наших классов. Закрытое поле context позволяет увидеть, как из IServiceCollection можно извлечь DbContext, который мы добавили туда, опять же, в классе Startup. Вернее в данном случае речь идет о создании экземпляра DbContext, ведь lifetime DbContext по умолчанию — Scope, но это уже отдельная тема. И да, в конструктор каждого DbContext автоматически будут передаваться аргументы такие, какие мы указали ASP однажды — в классе, в очередной раз, Startup. По сути поинт данного действия — один раз показать ASP как создавать экземпляры объектов и далее просто пользоваться ими. Либо один раз создать этот экземпляр (в случае со службами типа Singleton), после чего использовать его везде где нужно.

Таким образом микросервисная архитектура позволяет разделить программу на слабосвязанные кирпичики (связать которые можно также через внедрение зависимостей), а также внедрить зависимости и применить одну из основных особенностей ASP — инверсию управления.

20.12.2021 22:31  
vk.com/@65zio-mikroservisnaya-arhitektura-v-mvcnet