Create the interface that all tenant will implement

public interface IHasTenant

{

public string TenantName { get; set; }

}

Create entities to implement the interface

public class Habit : IHasTenant

{

public int Id { get; set; }

public string Name { get; set; } = default!;

public string Description { get; set; } = default!;

public string TenantName { get; set; } = default!;

}

Create section in the appsettings to hold the tenant settings

"TenantSettings": {

"DefaultConnectionString": "Server=.;Database=GoodHabitsDatabase;Trusted\_Connection=True;MultipleActiveResultSets=true",

"Tenants": [

{

"TenantName": "AscendTech",

"ConnectionString": "Server=.;Database=AscendTechGoodHabitsDatabase;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

{

"TenantName": "Bluewave",

"ConnectionString": "Server=.;Database=BluewaveGoodHabitsDatabase;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

{

"TenantName": "CloudSphere"

},

{

"TenantName": "Datastream"

}

]

}Create two classes in which these configuration will be read into;

Tenant

public class Tenant

{

public string? TenantName { get; set; }

public string? ConnectionString { get; set; }

}

And then TenantSettings

public class TenantSettings

{

public string? DefaultConnectionString { get; set; }

public List<Tenant>? Tenants { get; set; }

}

These two classes will be read and populated using the IOptions method during registration in the DI container using Configure method

Now we need a Tenant service that will give us a tenant as well as its connection string.

This class will be injected with

* IOptions<TenantSettings> -> so that it can have access to TenantSettings gotten from AppSettings.
* IhttpContextAccessor -> so that we could read request headers from the Httpcontext to see if we have a header with the name tenant

We start with the interface

public interface ITenantService

{

public string GetConnectionString();

public Tenant GetTenant();

}

Now the implementation

public class TenantService : ITenantService

{

private readonly TenantSettings \_tenantSettings;

private HttpContext \_httpContext;

private Tenant \_tenant;

public TenantService(IOptions<TenantSettings> tenantSettings, IHttpContextAccessor contextAccessor)

{

\_tenantSettings = tenantSettings.Value;

\_httpContext = contextAccessor.HttpContext!;

if (\_httpContext != null)

{

if (\_httpContext.Request.Headers.TryGetValue("tenant", out var tenantId))

{

SetTenant(tenantId!);

}

else

{

throw new Exception("Invalid Tenant!");

}

}

}

private void SetTenant(string tenantId)

{

\_tenant = \_tenantSettings!.Tenants.Where(a => a.TenantName == tenantId).FirstOrDefault();

if (\_tenant == null) throw new Exception("Invalid Tenant!");

if (string.IsNullOrEmpty(\_tenant.ConnectionString)) SetDefaultConnectionStringToCurrentTenant();

}

private void SetDefaultConnectionStringToCurrentTenant() =>

\_tenant.ConnectionString = \_tenantSettings.DefaultConnectionString;

public string GetConnectionString() => \_tenant?.ConnectionString!;

public Tenant GetTenant() => \_tenant;

}

**IServiceCollection configuration**

We add two extension methods to IServiceCollection

1.The first one when called and given a class(TenantSettings in our case), the iconfiguration class and a section name(a section in the configuration file) will read the section and populate the class

public static T GetOptions<T>(this IServiceCollection services, string sectionName,IConfiguration config) where T : new()

{

var options = new T();

config.GetSection(sectionName).Bind(options);

return options;

}

2.The second extension method will be used to add and migrate the database. So it will perform the following

* Call the first extention method and get the TenantSettings.
* Create a connectionstring using DefaultConnectionString of the TenantSettings
* Registers the dbcontext

Then for all Tenants found in the TenantSettings, it will

* Get their connectionstring(if you don’t have, it will use default from above), create a dbcontext from the iservicecollection and sets its connectionstring
* It will check if the database of the context has migrations and execute them.

The extension methods are below

public static class ServiceCollectionExtensions

{

public static IServiceCollection AddAndMigrateDatabases(this IServiceCollection services, IConfiguration config)

{

var options = services.GetOptions<TenantSettings>(nameof(TenantSettings),config);

var defaultConnectionString = options.DefaultConnectionString;

services.AddDbContext<GoodHabitsDbContext>(m => m.UseSqlServer(e => e.MigrationsAssembly(typeof(GoodHabitsDbContext).Assembly.FullName)));

var tenants = options.Tenants;

foreach (var tenant in tenants)

{

string connectionString;

if (string.IsNullOrEmpty(tenant.ConnectionString))

{

connectionString = defaultConnectionString;

}

else

{

connectionString = tenant.ConnectionString;

}

using var scope = services.BuildServiceProvider().CreateScope();

var dbContext = scope.ServiceProvider.GetRequiredService<GoodHabitsDbContext>();

dbContext.Database.SetConnectionString(connectionString);

if (dbContext.Database.GetMigrations().Count() > 0)

{

dbContext.Database.Migrate();

}

}

return services;

}

public static T GetOptions<T>(this IServiceCollection services, string sectionName,IConfiguration config) where T : new()

{

var options = new T();

config.GetSection(sectionName).Bind(options);

return options;

}

}

DbContext:

We create a dbcontext, making sure that

1. We inject it with TenantService so that we can get the tenant as well as his connection string to use and configure sqlserver
2. We also make sure that every entity that implements the IHasTenant interface should have a queryfilter:so that it is applied to every query of that entity type.
3. We can also seed the database here;NB:this seed data will go to all Databases;
4. We override the savechanges method and say that every entity that implements IHasTenant: if your state is either added or mordified, your Tenant Name should be The Name gotten from the injected TenantService

public class GoodHabitsDbContext : DbContext

{

private readonly ITenantService \_tenantService;

public GoodHabitsDbContext(DbContextOptions options, ITenantService service) : base(options) => \_tenantService = service;

public string TenantName { get => \_tenantService.GetTenant()?.TenantName ?? String.Empty; }

public DbSet<Habit>? Habits { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

var tenantConnectionString = \_tenantService.GetConnectionString();

if (!string.IsNullOrEmpty(tenantConnectionString))

{

optionsBuilder.UseSqlServer(\_tenantService.GetConnectionString());

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<Habit>().HasQueryFilter(a => a.TenantName == TenantName);

SeedData.Seed(modelBuilder);

}

public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = new CancellationToken())

{

ChangeTracker.Entries<IHasTenant>()

.Where(entry => entry.State == EntityState.Added || entry.State == EntityState.Modified)

.ToList()

.ForEach(entry => entry.Entity.TenantName = TenantName);

return await base.SaveChangesAsync(cancellationToken);

}

}

Finally we register services with the dependency injection container

builder.Services.AddHttpContextAccessor();

builder.Services.Configure<TenantSettings>(builder.Configuration.GetSection(nameof(TenantSettings)));

builder.Services.AddTransient<ITenantService,TenantService>();

builder.Services.AddAndMigrateDatabases(builder.Configuration);

Now we can create repositories for our entities eg Habit,register them with DI and use them.

**NB: The tenant name you pass in the requests headers is case Sensitive**