Prueba técnica TEKUS

Arquitectura:

Pasos en el desarrollo:

* Se crea repositorio en github con template de visual studio para que de una vez quede listo el archivo .gitignore con los valores apropiados
* Se clona proyecto localmente con git clone <https://github.com/juanidamato/pruebatecnica.git>
* Se crea solución en blanco de visual studio
* Se agrega proyecto TekusWeAPI a la solución (template .net core Web API)
* Se agrega proyecto TekusCore a la solución (template .net library)
* Se agrega proyecto TekusTests a la solución (template xUnit)
* Se agregan referencia de proyecto TekusWeAPI al proyecto TekusCore
* Se agregan referencia de proyecto TekusTests al proyecto TekusCore
* Se agregan algunos nugets al proyecto TekusWeAPI (ver archivo .csproj)
* Se agregan algunos nugets al proyecto TekusCore (ver archivo .csproj)
* Se agregan algunos nugets al proyecto TekusTests (ver archivo .csproj)
* Se configura archivo appsetting.json con conexión a base de datos y con parámetros de seguridad (EN LA VIDA REAL ESTOS PARAMETROS EN DESARROLLO SE GUARDAN EN UN ARCHIVO DE SECRETOS PARA DESARROLLO Y EN PRODUCCION SE PUEDEN OBTENER DE VARIABLES DE ENTORNO O DE UN KEYVAULT, NO SE DEBE USAR EL ARCHIVO appsettings.json porque quedarían publicados en el repositorio github estos datos)
* Se configura el archivo Program.cs del proyecto TekusWeAPI para la parte de generación y validación de tokens. (En un proyecto de la vida real sería mejor utilizar un servicio externo para la autenticación como Microsoft Business to Consumer aka B2C)
* Se crea el controlador de autenticación con unos modelos de entrada y salida y una utilidad para mapear los status codes de HTTP con los códigos de error de nuestras reglas de negocio BLL
* Se modifica el archivo Program.cs para configurar swagger de tal forma que permita utilizar Bearer tokens en los endpoints de las API
* Para generar un token se utiliza el endpoint api/auth/token por el método POST con usuario “admin“ y clave “admin” (cualquier otro valor generar código HTTP 401)
* El valor del access token generado se copia en el clipboard y se pega en swagger para ser utilizado en los otros endpoints seguros (Anteponer la palabra Bearer y un espacio en blanco antes del token jwt). El token generado también tiene un claim de scope "api.access" el cual debe ser validado en los métodos del los controllers o por medio de un custom middleware (en mi caso utilizo el atributo RequiredScope )
* Se configura en el Progam.cs la inyección de dependencias para que se haga un Discovery automatico de las clases que tienen el patrón de “Mediator” para implementar el patrón de CQRS , las clases de validators que utilizan la librería de fluentValidation y los mappers de automapper
* Tambien se configuran las inyecciones de dependencias y sus correspondientes scopes de las clases propias de la lógica del negocio de la aplicación y de todas las abstracciones que se crean con sus correspondientes implementaciones.
* Se crea base de datos “dbTekus” con collation Latin1\_General\_CI\_AI. Se escoge este collation porque acepta caracteres “especiales” y porque no es sensible a comparaciones por letras mayúsculas y minúsculas y tampoco a aspectos de acento como por ejemplo comparar la palabra “producción” y “produccion” (note la tilde)
* Se crea la tabla Providers (en plural puesto que se corre el riesgo Microsoft agregue como palabra reservada de T-SQL la palabra Provider)
* Se crea la tabla Services (también en plural por las mismas razones)
* Se crea la tabla relacional ProviderService que relaciona para un porvider cuales servicios se prestan (Aca hay dos tendencias en la industria una es crear esta tabla con una llave única autonumerica y un índice único de proveedor,servicio,país y la otra tendencia es que esta tabla no tenga este campo autonumerico y simplemente la llave sea una llave compuesta, en lo personal me gusta la primera tendencia)
* Tambien se crean dos llaves foráneas entre la tabla ProviderService y las tablas de proveedores y servicios (Por performance también se pueden crear índices sobre estos campos aplica si hay muchos registros en esta tabla para que al chequear la integridad relacional le ayudemos al motor)
* Para los campos personalizados del provedor se tienen otras 3 tablas (la primera define todos los posibles campos personalizados, la segunda define para cada provider cuales campos personalizados se utilizaran y la tercera tabla contiene los valores de estos campos personalizados para cada provider) El campo FieldType de la primera tabla para este ejercicio tiene un valor fijo a texto pero puede extenderse el concepto a una fecha, un número, etc.
* Se procede a crear un primer endpoint para listar los proveedores (el objetivo de este endpoint es mostrar la información básica de todos los proveedores en una tabla la cual puede ser ordenada por algunos campos y también se plantea que tenga paginación es decir que la vista muestra resultados parciales). Se asume también que en esta del frontend tiene viculos o botones que permiten las operaciones de inserción, actualización y borrado para administrar los proveedores