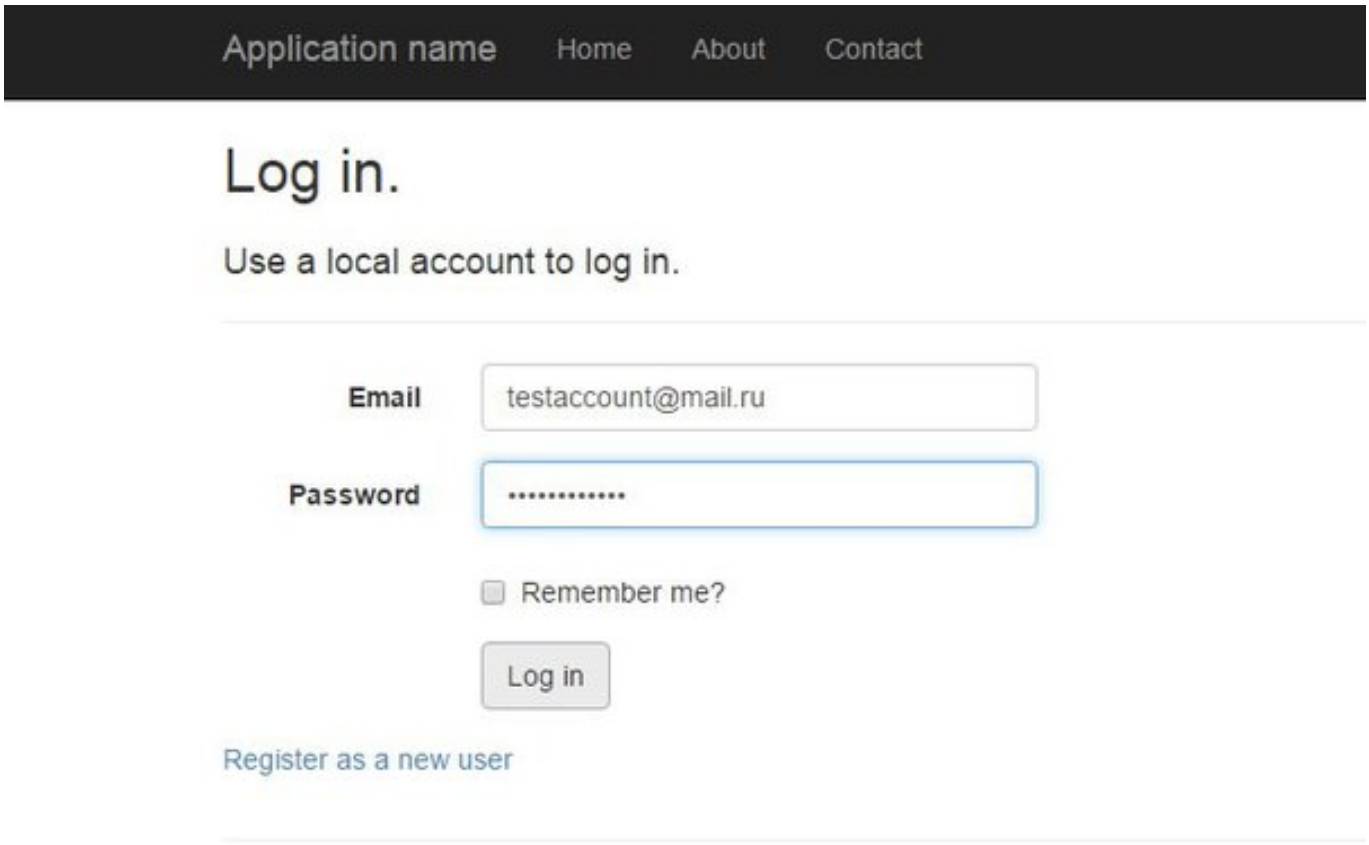
Article
Maxim Yerokhin · Sep 21, 2016 7m read

ASP.NET Identity Caché Provider — working with Identity via InterSystems Caché

Imagine that your .NET project uses the Caché DBMS and you need a fully-functional and reliable authorization system. Writing such a system from scratch would not make much sense, and you will clearly want to use something that already exists in .NET, e.g. ASP.NET Identity. By default, however, this framework supports only its native DBMS – MS SQL. Our task was to create an adaptor that would let us quickly and easily port Identity to the InterSystems Caché DBMS. This work resulted in creation of the ASP.NET Identity Caché Provider.



MSSQL is the default data provider for ASP.NET Identity, but since Identity ' s authorization system can interact with any other relational DBMS, we implemented this functionality for InterSystems Caché.

The goal of the ASP.NET Identity Caché Provider project was to implement a Caché data provider that would work with ASP.NET Identity. The main task was to store and provide access to such tables as AspNetRoles, AspNetUserClaims, AspNetUserLogins, AspNetUserRoles and AspNetUsers without breaking the standard workflows involving these tables.

Let ' s take a look at the implementation of the Caché data provider for ASP.NET Identity. It had two phases:

- Implementation of data storage classes (that will be responsible for storing state data) and the *IdentityDbContext* class that encapsulates all low-level logic for interaction with the data storage. We also

implemented the IdentityDbInitializer class that adapts the Caché database for working with Identity.
- Implementation of the *UserStore* and *RoleStore* classes (along with integration tests). A demo project.

During the first stage, the following classes were implemented:

- *IdentityUser* — implementation of the IUser interface.
- *IdentityUserRole* — an associative entity for the User–Role pair.
- *IdentityUserLogin* — user login data.

Extendable version of the UserLoginInfo class.

- IdentityUserClaim —information about the user ' s claims.
- IdentityDbContext<TUser, TRole, TKey, TUserLogin, TUserRole, TUserClaim> — context of the Entity Framework database.

Let ' s take a look at the *IdentityUser* entity more detailed. It is a storage for users, roles, logins, claims and user-role relations. Below there is an example of a regular and generalized variant of *IdentityUser*.

```
namespace InterSystems.AspNet.Identity.Cache
{
    /// <summary>
    /// IUser implementation
    /// </summary>
    public class IdentityUser : IdentityUser<string, IdentityUserLogin, IdentityUserR
ole, IdentityUserClaim>, IUser
    {
        /// <summary>
        /// Constructor which creates a new Guid for the Id
        /// </summary>
        public IdentityUser()
        {
            Id = Guid.NewGuid().ToString();
        }

        /// <summary>
        /// Constructor that takes a userName
        /// </summary>
        /// <param name="userName"></param>
        public IdentityUser(string userName)
            : this()
        {
            UserName = userName;
        }
    }

    /// <summary>
    /// IUser implementation
    /// </summary>
    /// <typeparam name="TKey"></typeparam>
    /// <typeparam name="TLogin"></typeparam>
```

```
    /// <typeparam name="TRole"></typeparam>
    /// <typeparam name="TClaim"></typeparam>
    public class IdentityUser<TKey, TLogin, TRole, TClaim> : IUser<TKey>
        where TLogin : IdentityUserLogin<TKey>
        where TRole : IdentityUserRole<TKey>
        where TClaim : IdentityUserClaim<TKey>
    {
        /// <summary>
        ///     Constructor
        /// </summary>
        public IdentityUser()
        {
            Claims = new List<TClaim>();
            Roles = new List<TRole>();
            Logins = new List<TLogin>();
        }

        /// <summary>
        /// Email
        /// </summary>
        public virtual string Email { get; set; }
```

Special objects called Roles are intended for access rights restrictions in Identity. A role in the configuration can correspond to job positions or types of activities of various user groups.

```
namespace InterSystems.AspNet.Identity.Cache
{
    /// <summary>
    /// EntityType that represents a user belonging to a role
    /// </summary>
    public class IdentityUserRole : IdentityUserRole<string>
    {
    }


    /// <summary>
    /// EntityType that represents a user belonging to a role
    /// </summary>
    /// <typeparam name="TKey"></typeparam>
    public class IdentityUserRole<TKey>
    {
        /// <summary>
        /// UserId for the user that is in the role
        /// </summary>
        public virtual TKey UserId { get; set; }

        /// <summary>
        /// RoleId for the role
        /// </summary>
        public virtual TKey RoleId { get; set; }
    }
}
```

IdentityDbContext is an instance that encapsulates the creation of a connection, loading of entities from a database, validation of user ' s objects conformity to the structure of associated tables and field values. Let ' s use the *OnModelCreating* as an example – this method validates tables according to Identity requirements.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
        // Mapping and configuring identity entities according to the Cache tables
        var user = modelBuilder.Entity<TUser>()
           .ToTable("AspNetUsers");
            user.HasMany(u => u.Roles).WithRequired().HasForeignKey(ur => ur.UserId);
            user.HasMany(u => u.Claims).WithRequired().HasForeignKey(uc => uc.UserId)
;
            user.HasMany(u => u.Logins).WithRequired().HasForeignKey(ul => ul.UserId)
;
            user.Property(u => u.UserName)
                .IsRequired()
                .HasMaxLength(256)
                .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute(
"UserNameIndex") { IsUnique = true }));

            user.Property(u => u.Email).HasMaxLength(256);
            modelBuilder.Entity<TUserRole>()
                .HasKey(r => new { r.UserId, r.RoleId })
                .ToTable("AspNetUserRoles");

            modelBuilder.Entity<TUserLogin>()
                .HasKey(l => new { l.LoginProvider, l.ProviderKey, l.UserId })
                .ToTable("AspNetUserLogins");

            modelBuilder.Entity<TUserClaim>()
                .ToTable("AspNetUserClaims");

            var role = modelBuilder.Entity<TRole>()
                .ToTable("AspNetRoles");

            role.Property(r => r.Name)
                .IsRequired()
                .HasMaxLength(256)
                .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute(
"RoleNameIndex") { IsUnique = true }));

            role.HasMany(r => r.Users).WithRequired().HasForeignKey(ur => ur.RoleId);
}
```

DbModelBuilder is used for comparing CLR classes with the database schema. This code-oriented approach to build an EDM model is called Code First. *DbModelBuilder* is typically used for configuring the model by means of redefining *OnModelCreating*(*DbModelBuilder*). However, *DbModelBuilder* can also be used independently from *DbContext* for model creation and subsequent design of *DbContext* or *ObjectContext*.

The *IdentityDbInitializer* class prepares the Caché database for using *Identity*.

```
public void InitializeDatabase(DbContext context)
{
     using (var connection = BuildConnection(context))
     {
           var tables = GetExistingTables(connection);
           CreateTableIfNotExists(tables, AspNetUsers, connection);
           CreateTableIfNotExists(tables, AspNetRoles, connection);
           CreateTableIfNotExists(tables, AspNetUserRoles, connection);
           CreateTableIfNotExists(tables, AspNetUserClaims, connection);
           CreateTableIfNotExists(tables, AspNetUserLogins, connection);
           CreateIndexesIfNotExist(connection);
      }
}
```

CreateTableIfNotExists method creates the necessary tables if they don't exist. Table existence checks are performed by running a query against the **Cache** – **Dictionary.CompiledClass** table that stores information about existing tables. If the table doesn't exist, it will be created.

On the second stage, *IdentityUserStore* and *IdentityRoleStore* instances were created. They encapsulate the logic of adding, editing and removing users and roles. These entities required 100% unit-test coverage.

Let's draw the bottom line: we created a data provider that allows the Caché DBMS to work with Entity Framework within the context of the ASP.NET Identity technology. The app is packed into a separate Nuget package, so if you need to work with the Caché DBMS and use standard Microsoft authorization, all you need to do is to add the Identity Caché Provider build into your project via Nuget Package Manager.

The source code of the project, along with samples and documentation, is available on GitHub.

#.NET #Authentication #Access control #Caché