
Blazor Boilerplate

Release 4.0.0

Dec 26, 2022

1	Blazor Server	3
1.1	Pros	3
1.2	Cons	3
2	Blazor WebAssembly	5
2.1	Pros	5
2.2	Cons	5
2.2.1	Architecture	7
2.2.2	Security	7
2.2.3	Breeze Sharp with Blazor	9
2.2.4	Entity Framework Core	10
2.2.5	Localization	11
2.2.6	Deploy with Terraform	12
2.2.7	Hosting Blazor Boilerplate 2.0.0 on Azure	13
2.2.8	Dual mode Blazor	23
2.2.9	MultiTenancy	24
2.2.10	IdentityServer4	25
2.2.11	Source Generators	26



Blazor Boilerplate is a starter template for [Blazor web apps](#).

With Blazor Boilerplate you can easily switch between the two modes of Blazor.

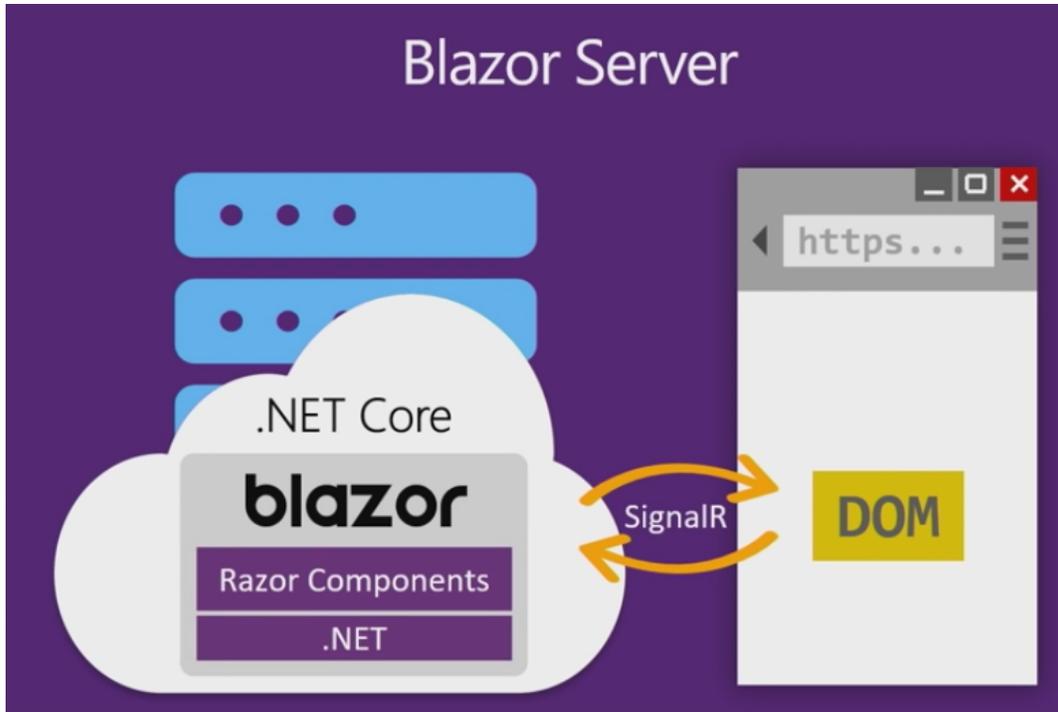
From Microsoft documentation [ASP.NET Core Blazor hosting models](#).

1.1 Pros

- Download size is significantly smaller than a Blazor WebAssembly app, and the app loads much faster.
- The app takes full advantage of server capabilities, including use of any .NET Core compatible APIs.
- .NET Core on the server is used to run the app, so existing .NET tooling, such as debugging, works as expected.
- Thin clients are supported. For example, Blazor Server apps work with browsers that don't support WebAssembly and on resource-constrained devices.
- The app's .NET/C# code base, including the app's component code, isn't served to clients.

1.2 Cons

- Higher latency usually exists. Every user interaction involves a network hop.
- There's no offline support. If the client connection fails, the app stops working.
- Scalability is challenging for apps with many users. The server must manage multiple client connections and handle client state.
- An ASP.NET Core server is required to serve the app. Serverless deployment scenarios aren't possible (for example, serving the app from a CDN).

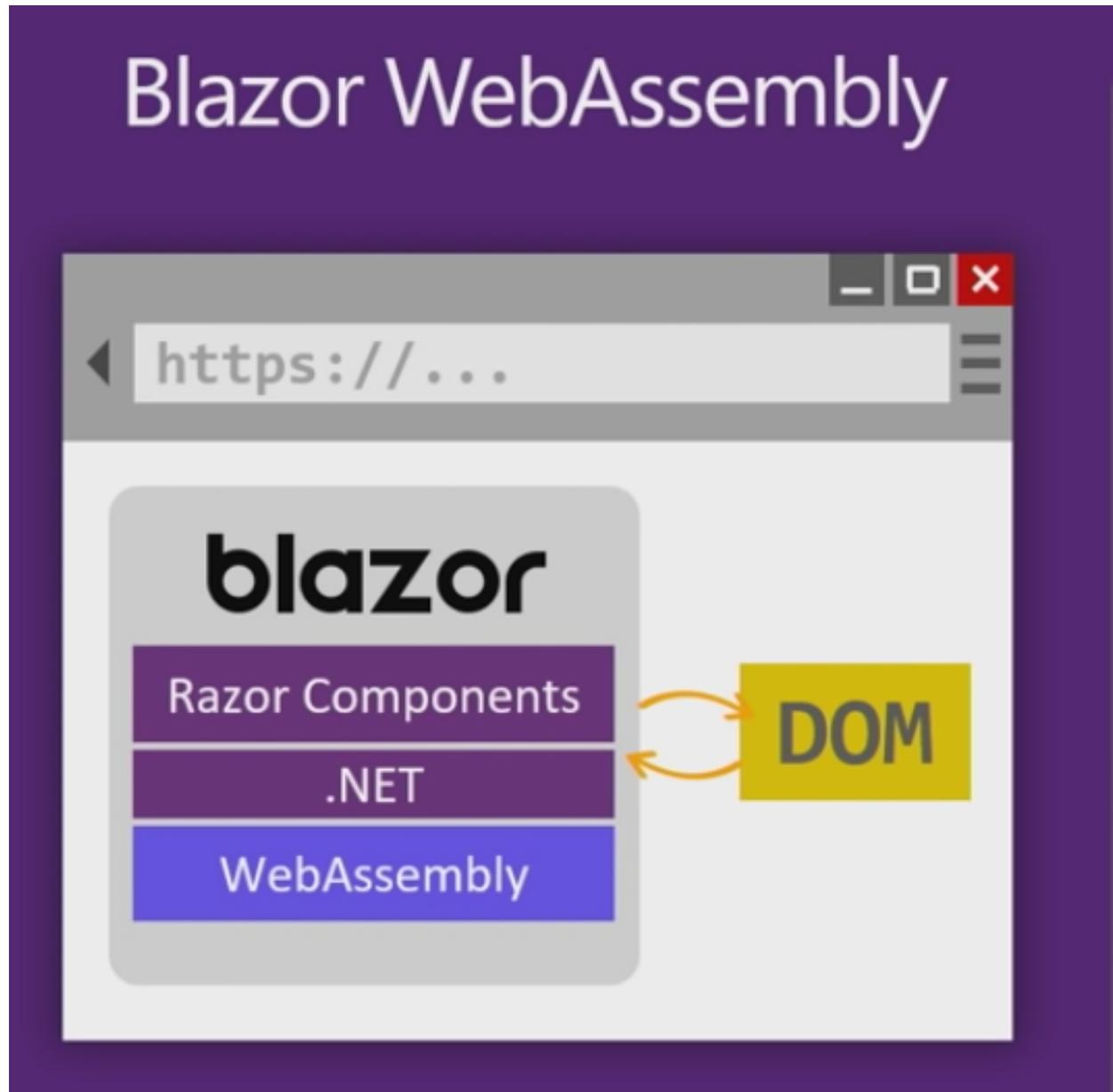


2.1 Pros

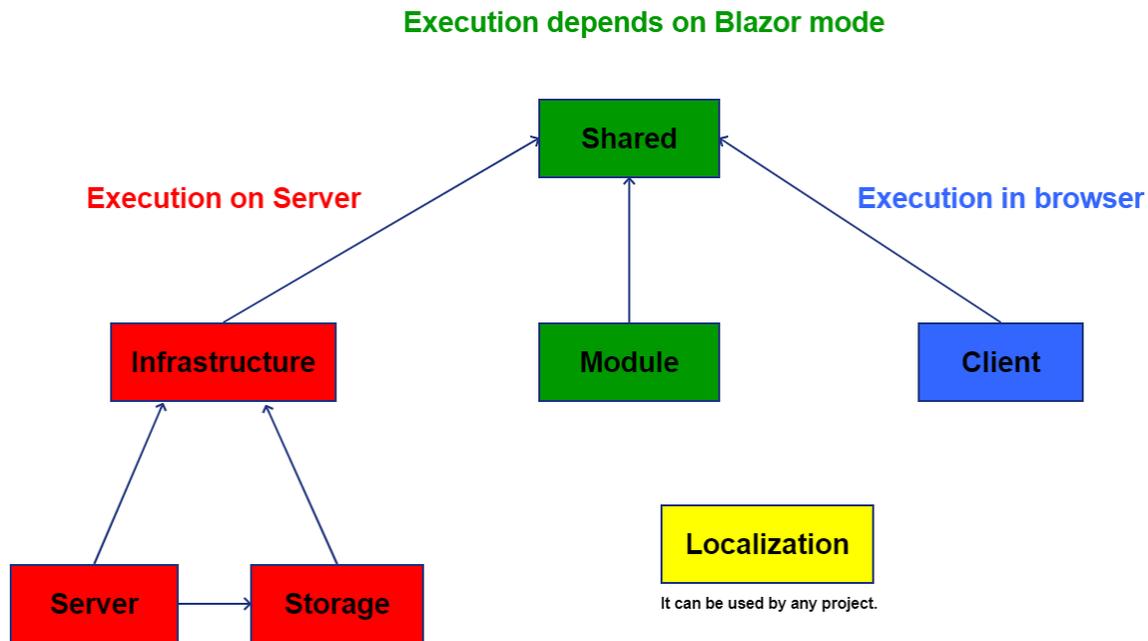
- There's no .NET server-side dependency. The app is fully functioning after it's downloaded to the client.
- Client resources and capabilities are fully leveraged.
- Work is offloaded from the server to the client.
- An ASP.NET Core web server isn't required to host the app. Serverless deployment scenarios are possible (for example, serving the app from a CDN).

2.2 Cons

- The app is restricted to the capabilities of the browser.
- Capable client hardware and software (for example, WebAssembly support) is required.
- Download size is larger, and apps take longer to load.
- .NET runtime and tooling support is less mature. For example, limitations exist in .NET Standard support and debugging.



2.2.1 Architecture



The diagram shows the dependencies of the main projects. Every project with text to localize depends on **Localization** project.

Client project is used only with Blazor WebAssembly, so it runs on the browser. It initializes the WebAssembly startup. If you only use Blazor Server, remove this project from solution.

Server project manages the main services: authentication, authorization, API, etc.

Storage project manages the persistence of models in a database with [Entity Framework Core](#).

Infrastructure project contains interfaces and models in common among projects running on server.

Shared project contains base interfaces and models in common among projects running on server or client.

Module projects contains UI, services, etc. that define your web application.

2.2.2 Security

Administrator

In BlazorBoilerplate.Constants project change Administrator name to something less obvious than “admin”.

```

namespace BlazorBoilerplate.Constants
{
    public static class DefaultUserNames
    {
        public const string Administrator = "iamtheboss";
        public const string User = "user";
    }
}
  
```

In the same project reinforce password policy.

```
namespace BlazorBoilerplate.Constants
{
    public static class PasswordPolicy
    {
        public const bool RequireDigit = true;
        public const int RequiredLength = 8;
        public const bool RequireNonAlphanumeric = true;
        public const bool RequireUppercase = true;
        public const bool RequireLowercase = true;
    }
}
```

In DatabaseInitializer in BlazorBoilerplate.Storage project change Administrator password “admin123” to satisfy new policy.

```
public async Task EnsureAdminIdentitiesAsync()
{
    await EnsureRoleAsync(DefaultRoleNames.Administrator, _entityPermissions.
↵ GetAllPermissionValues());
    await CreateUserAsync(DefaultUserNames.Administrator, "X!PvG5+@", "Admin", "Blazor
↵", "admin@blazorboilerplate.com", "+1 (123) 456-7890", new string[] {
↵ DefaultRoleNames.Administrator });
}
```

API endpoints

Imagine a malicious user writing or using a tool to call directly your API endpoints bypassing your UI, are your API endpoints protected by the right policy?

```
[HttpGet]
[Authorize(Policies.IsAdmin)]
public IQueryable<ApplicationUser> Users()
{
    return persistenceManager.GetEntities<ApplicationUser>().AsNoTracking().Include(i
↵ => i.UserRoles).ThenInclude(i => i.Role).OrderBy(i => i.UserName);
}
```

What about the Breeze SaveChanges endpoint?

```
[AllowAnonymous]
[HttpPost]
public SaveResult SaveChanges([FromBody] JObject saveBundle)
...
```

First of all remove [AllowAnonymous] attribute. An authenticated malicious user could post a proper json to update some EF entities. To avoid this, decorate your EF entities with proper Permission attribute. E.g.

```
namespace BlazorBoilerplate.Infrastructure.Storage.DataModels
{
    [Permissions(Actions.Delete)]
    public partial class Todo : IAuditable, ISoftDelete
    ...
}
```

Add the entity permissions to a role and put a user in this role.

2.2.3 Breeze Sharp with Blazor

Note: What you will read here are my personal opinions. I have been using Breeze from years for both web and desktop applications and I have no connection with Breeze team.

Giovanni Quarella

Recommended **CRUD API design** is too verbose and boring.

For each CRUD action Create, Read, Update and Delete you have to add a method to your API controller. Also you have to repeat the same pattern for every entity of your application domain.

In real use cases you cannot perform also single actions on entities, because changes have to happen at the same time in a transaction. So you have to write other code. . .

Do you know good coders are lazy? Do you know the DRY principle?

The solution is a [Breeze Controller](#).

As you can read in the official documentation: *“One controller to rule them all . . . ”*.

Breeze Server

In this project the [Breeze Controller](#) is [ApplicationController](#) and the [EFPersistenceManager](#) is [ApplicationPersistenceManager](#).

The [ApplicationController](#) has no **Authorize** attribute, because to keep policy management with Breeze, I implemented a simple custom policy management in [ApplicationPersistenceManager](#).

First of all you can mark the EF entities with [Permissions](#) attribute which takes [CRUD Actions flag enum](#) as parameter. Examples:

```
[Permissions(Actions.Delete)]
public partial class Todo
{
    ...
}

[Permissions(Actions.Create | Actions.Update)]
public partial class Todo
{
    ...
}

[Permissions(Actions.CRUD)]
public partial class Todo
{
    ...
}
```

In [BeforeSaveEntities](#) method of [ApplicationPersistenceManager](#) the actions of the entities to be saved are compared with the claims of permission type of the current user. If a policy is violated an [EntityErrorsException](#) is thrown.

To check **Actions.Read** in [ApplicationController](#) you have to access EF DbSet with **GetEntities** method of [ApplicationPersistenceManager](#).

Breeze.Sharp (Breeze C# client)

To access Breeze Controller you have to use `Breeze.Sharp`. In Blazor BlazorBoilerplate `ApiClient` is what you need to query `ApplicationController`. You can inject `IApiClient` in every Razor page where entities are requested.

The `Breeze.Sharp` entities are the so called DTO and you should create one for every entities of your Entity Framework context in `EF` entities. To avoid repetitive tasks, I created `EntityGenerator` based on `Source Generators`. Every time you update `EF` entities, rebuild `BlazorBoilerplate.Shared` project to have Breeze entities automatically generated with namespace `BlazorBoilerplate.Shared.Dto.Db`.

Notes

At the moment `Todo` is the only entity used to demonstrate `Breeze.Sharp` capabilities. In fact you can extend the use to all other entities, but administrative entities (users, roles, etc.) are managed by others libraries like `ASP.NET Core Identity` and `IdentityServer4`, so I preferred to keep the dedicated `AdminController`.

If you think that is all wonderful, here it comes the **drawback**.

The used of C# as a replacement of javascript is something very new: it is the Blazor revolution. So till now `Breeze.Sharp` has been used a little only on desktop application. In fact the most used Breeze client is the javascript client `BreezeJS` and `Breeze` itself is not .NET centric.

For this reason `breeze.sharp` library is rarely updated, so to make `BlazorBoilerplate` working with Breeze I used my fork `GioviQ/breeze.sharp` where I fixed some issues. You can find the package in <https://www.nuget.org/packages/Breeze.Sharp.Standard.Fork/>.

2.2.4 Entity Framework Core

Getting started with EF Core (code first) reading this [tutorial](#).

Migrations

You can perform Entity Framework Core migrations directly from Visual Studio with [Package Manager Console](#) or with [command-line interface](#).

Here some tips to manage migrations in Blazor Boilerplate with command-line interface (CLI).

1. Make sure you have installed CLI; from the command line execute:

```
dotnet tool install --global dotnet-ef
```

or to update

```
dotnet tool update --global dotnet-ef
```

2. Keep in mind that every `DbContext` has its own migrations.

The main `DbContext` is `ApplicationDbContext`. `PersistedGrantDbContext` and `ConfigurationDbContext` depend on `IdentityServer4`; if new versions have changed one or both db contexts, you have to add migrations. The migrations are in `BlazorBoilerplate.Storage`, so you have to open your shell in this path. Then you have to specify the project where the db contexts are added to the services, because `dotnet ef` has to know how to find and instantiate them. In our case the so called startup project is `BlazorBoilerplate.Server`. So the commands are:

```
dotnet ef --startup-project ../BlazorBoilerplate.Server/ migrations add 4Preview3 -c_
↳PersistedGrantDbContext --verbose --no-build --configuration Debug
```

```
dotnet ef --startup-project ../BlazorBoilerplate.Server/ migrations add 4Preview3 -c_
↳ConfigurationDbContext --verbose --no-build --configuration Debug
```

Without the **--no-build** parameter, **dotnet** rebuilds the startup project, but if you have just built it with Visual Studio, it is just a waste of time. Also the **dotnet** build is not the Visual Studio build, so to avoid issue, use this procedure.

The name of migration, in this case *4Preview3*, is only descriptive and it does not need to be unique, because automatically the name is prepended with date time.

If the command is successful, **[datetime]_migration-name.cs** is added to the solution. Also the **[db-context-name]ModelSnapshot.cs** is updated to reflect the db changes in case a new db has to be created.

When you run the project, the migrations are applied to the database (in our case we have only one db). The db table **__EFMigrationsHistory** keeps track of applied migrations without information about the related db context. To get this information use the following command; for example for ConfigurationDbContext:

```
dotnet ef --startup-project ../BlazorBoilerplate.Server/ migrations list -c_
↳ConfigurationDbContext --verbose --no-build --configuration Debug
```

You can also update the database, without running the project with the following command:

```
dotnet ef --startup-project ../BlazorBoilerplate.Server/ database update 4Preview3 -c_
↳ConfigurationDbContext --verbose --no-build --configuration Debug
```

If you specify a previous migration, you can revert the db changes to that migration.

Warning

The migrations are not smart, when you have an existing db with populated tables. If migrations add keys and/or unique indexes or something else violating referential integrity, you have to manually modify **[datetime]_migration-name.cs** for example to add some SQL to fix the errors. E.g. **migrationBuilder.Sql("UPDATE AspNetUserLogins SET Id=NEWID() WHERE Id=");** to add unique values to a new field before setting as a new primary key.

Shadow Properties vs Source Generator

Always keeping in mind the DRY principle, it is boring implementing audit information and adding the same properties **CreatedOn**, **ModifiedOn**, **CreatedBy** and **ModifiedBy** to all entities.

Some articles teach you to use **Shadow Properties** to add audit information, but this is not the right solution, if you want expose these properties on the mapped entity types and use them e.g. in UI.

A solution is using **Source Generator**. **AuditableGenerator** generates for every class implementing **IAuditable** the above properties. Remember all classes to be extended by Source Generator have to be **partial**.

2.2.5 Localization

Localization has moved from Resource based localization to Database based localization, so translation are no more in dll satellite libraries, but in table **LocalizationRecords**. Look at **LocalizationDbContext**.

The localization code is in **BlazorBoilerplate.Shared.Localizer** project. The supported cultures are defined in **Settings.cs**.

At this time **Data Annotations** do not support **IStringLocalizer<>**, so to localize validation error messages, we have to use **Blazored.FluentValidation**.

Po files

This project adopts the [PO file format](#) as standard translations files. In the admin section you can edit translations and import and export PO files. There are a lot of free and paid tools and online services to manage these files. E.g.:

Poedit

Eazy Po

Crowdin (PO file format support)

To manage PO files BlazorBoilerplate uses [Karambolo.PO](#) library. So to handle plurals the Karambolo.PO syntax is used like in the example below.

Everywhere you need localized text, inject **IStringLocalizer<Global>** and look how it is used throughout the code.

```
L["MsgId usually in english"]  
  
L["Email {0} is not valid", email]  
  
L["One item found", Plural.From("{0} items found", totalItemsCount)]
```

2.2.6 Deploy with Terraform

Terraform Overview

- Use terraform to deploy to nearly any hosting environment
- Terraform is open source <https://github.com/hashicorp/terraform>
- For more information go to <https://www.terraform.io/>

Pass

1. Install and initialize **pass**.
 - pass makes managing these deployment parameters extremely easy
 - For instructions to install and initialize go to: <https://www.passwordstore.org/>
 - When you export your environment variables via export TF_VAR_variable_name, the variables get pulled in during terraform plan and terraform apply

Terraform for AWS

1. Once **pass** is installed and initialized do the following to store the environment variables in pass

```
pass insert access_key  
pass insert secret_key  
pass insert aws_key_name  
pass insert docker_blazorboilerplate_image  
pass insert docker_sqlserver_image  
pass insert sa_password  
pass insert cert_password  
pass insert ASPNETCORE_ENVIRONMENT  
pass insert Serilog_MinimumLevel_Default
```

2. Set terraform deployment environment variables for use in terraform:

```
export TF_VAR_access_key=$(pass access_key)
export TF_VAR_secret_key=$(pass secret_key)
export TF_VAR_aws_key_name=$(pass aws_key_name)
export TF_VAR_docker_blazorboilerplate_image=$(pass docker_blazorboilerplate_image)
export TF_VAR_docker_sqlserver_image=$(pass docker_sqlserver_image)
export TF_VAR_sa_password=$(pass sa_password)
export TF_VAR_cert_password=$(pass cert_password)
export TF_VAR_ASPNETCORE_ENVIRONMENT=$(pass ASPNETCORE_ENVIRONMENT)
export TF_VAR_Serilog_MinimumLevel_Default=$(pass Serilog_MinimumLevel_Default)
```

In Windows use setx instead. e.g.
setx TF_VAR_access_key=\$(pass access_key)

3. Install Terraform at <https://learn.hashicorp.com/tutorials/terraform/install-cli?in=terraform/aws-get-started>

- follow instructions **on** page

4. Deploy using Terraform

```
cd ./src/Utils/Terraform/AWS
terraform init
terraform plan
terraform apply
```

Terraform for Azure (Todo)

1. Once **pass** is installed and initialized do the following to store the environment variables in pass

```
todo:
pass insert param1
pass insert param2
```

2. Set terraform deployment environment variables for use in terraform:

```
todo:
export TF_VAR_param1=$(pass param1)
export TF_VAR_param2=$(pass param2)
```

In Windows use setx instead. e.g.
setx TF_VAR_param1=\$(pass param1)

3. Install Terraform at <https://learn.hashicorp.com/tutorials/terraform/install-cli?in=terraform/azure-get-started>

- follow instructions **on** page

4. Deploy using Terraform (Todo)

```
cd ./src/Utils/Terraform/Azure
terraform init
terraform plan
terraform apply
```

2.2.7 Hosting Blazor Boilerplate 2.0.0 on Azure

(Note: This document builds upon, and credits, an earlier pdf guide produced by @soomon).

This guide produces a working **demonstration** installation of Blazor Boilerplate on Azure App Services. Various configuration settings, security and design considerations **must be evaluated** before moving into production.

This guide is tested/working as at **June 2021**.

Preamble

Please be aware that hosting on Microsoft Azure will cost money. This tutorial assumes that you already have a working Azure subscription.

In this example, Blazor Boilerplate is being hosted using App Services and a managed SQL database.

The certificate for Identity Server has been stored in Azure Key Vault.

Visual Studio is being used to deploy the app to Azure. Version 16.10.0 was used in this example.

Ensure that you choose the same subscription, region and resource group for all resources created in this tutorial. Some Azure resources cannot see each other across regions.

All URIs, resources, usernames and passwords shown have been deleted prior to publishing.

Configure Azure Environment

Create the Azure Web App

Using either the Azure Portal App (<https://portal.azure.com/App/Download?acceptLicense=true>) or the portal web interface, create a new Web App and configure it as follows:

The screenshot shows the 'Create Web App' configuration page in the Azure Portal. The page is titled 'Create Web App' and has a breadcrumb 'Home > Create a resource >'. Below the title are tabs for 'Basics', 'Deployment (Preview)', 'Monitoring', 'Tags', and 'Review + create'. The 'Basics' tab is selected. The page contains several sections for configuring the web app:

- Project Details:** A message instructs the user to select a subscription to manage resources. Below this are two dropdown menus: 'Subscription' (set to 'Pay-As-You-Go') and 'Resource Group' (set to '(New) blazor_boilerplate_demo'). A 'Create new' link is visible below the Resource Group dropdown.
- Instance Details:** This section includes:
 - 'Name': 'blazor-boilerplate' (with '.azurewebsites.net' as a suffix).
 - 'Publish': Radio buttons for 'Code' (selected) and 'Docker Container'.
 - 'Runtime stack': '.NET 5'.
 - 'Operating System': Radio buttons for 'Linux' and 'Windows' (selected).
 - 'Region': 'Australia Southeast'. A note below says 'Not finding your App Service Plan? Try a different region.'
- App Service Plan:** A message explains that the pricing tier determines location, features, cost, and compute resources. Below this is a dropdown menu for 'Windows Plan (Australia Southeast)' set to '(New) blazor-boilerplate-app-service-plan'. A 'Create new' link is visible below.
- Skus and size:** The selected option is 'Free F1' with the description 'Shared infrastructure, 1 GB memory'. A 'Change size' link is visible below.

Resource Group: (create new) blazor_boilerplate_demo_rg

Instance Name: blazor-boilerplate *(.azurewebsites.net)

Publish: Code

Stack: .NET 5

OS: Windows

Region: (whatever works for you)

App Service Plan: (create new) blazor-boilerplate-app-service-plan. Select the Dev/Free F1 tier for demo purposes.

** The instance name must be globally unique. It is created in the azurewebsites.net domain.

Review + Create the new web app, then open the resource to continue with configuration.

Configure the Azure Web App

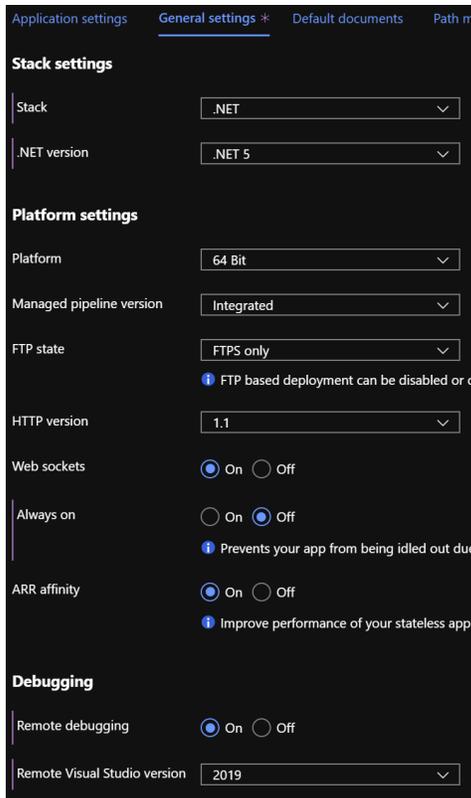
The screenshot shows the Azure Portal interface for a web app named 'blazor-boilerplate'. The left sidebar contains navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Events (preview), Deployment, Quickstart, Deployment slots, Deployment Center, Settings, and Configuration. The main content area shows the 'Essentials' section with the following details:

Resource group (change)	: blazor_boilerplate_demo	URL	: https://blazor-boilerplate.azurewebsites.net
Status	: Running	Health Check	: Not Configured
Location	: Australia Southeast	App Service Plan	: blazor-boilerplate-app-service-plan (F1: Free)
Subscription (change)	: Pay-As-You-Go	FTP/deployment username	: No FTP/deployment user set
Subscription ID	:	FTP hostname	:
Tags (change)	: Click here to add tags	FTPS hostname	:

Below the Essentials section, there are three informational boxes:

- Diagnose and solve problems:** Our self-service diagnostic and troubleshooting experience helps you identify and resolve issues with your web app.
- Application Insights:** Application Insights helps you detect and diagnose quality issues in your apps, and helps you understand what your users actually do with it.
- App Service Advisor:** App Service Advisor provides insights for improving app experience on the App Service platform. Recommendations are sorted by freshness, priority and impact to your app.

- Select **Configuration** and **General Settings** and configure as follows:



Stack: .NET

.NET Version: .NET 5**

Platform: 64 Bit

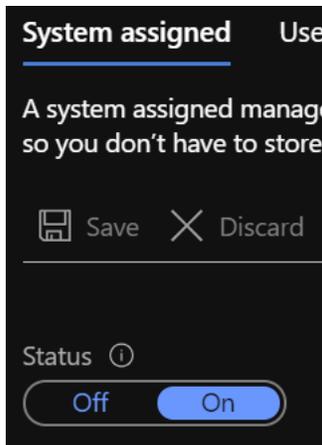
Pipeline: Integrated

Web Sockets: On - * Important

** Currently there is a bug in the portal that causes .NET version to display .NET Core (3.1/2.1) whenever you return to the general settings page. This is OK, it doesn't affect our demo.

Save these general settings and exit configuration.

- b. Select **Identity** and confirm that that **System Assigned** identity is turned **on**.



c. Configure **Custom Domains** and **TLS/SSL Settings**

By default, the new web app is accessible via `http://blazor-boilerplate.azurewebsites.net`. To enable various security features in Blazor Boilerplate to function in a hosted environment it must be secured with a certificate.

Either:

- secure the default URL (in the `azurewebsites.net` domain)

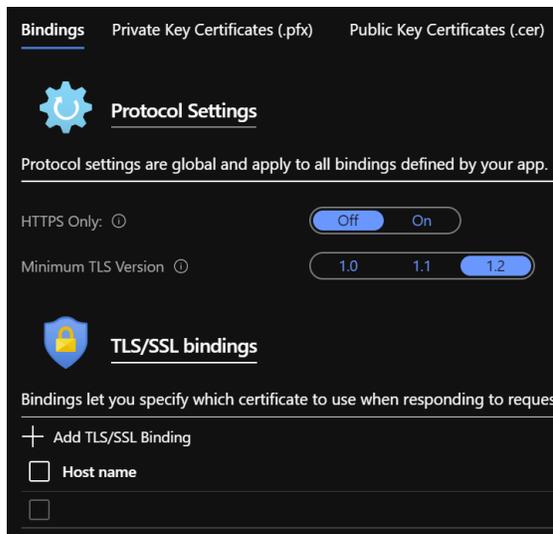
Or:

- secure your own host in a custom domain you control (e.g., `http://blazor-boilerplate.demodomain.com`)

If you are using a custom domain, select **Custom Domains** and add and verify a new custom domain. Note: you will need access to DNS host records for your chosen domain to verify it using the custom domain verification ID. In this example we have added `blazor-boilerplate.demodomain.com` as a custom domain.

Create a Private Key Certificate which will be used for TLS/SSL binding, using the hostname you have chosen, in this example `blazor-boilerplate.demodomain.com`. Use a development certificate like `AuthKey.pfx`, create a self-signed certificate or use a free service like `letsencrypt.org`. Make sure you mark the private key as exportable when you create the certificate.

Select **TLS/SSL Settings** and upload the `.pfx` certificate, then under **Bindings** add a binding to your chosen host name.

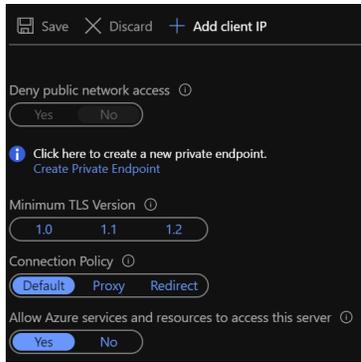


- d. Finally, select **Overview** and download the publish profile that you will later import into Visual Studio.

Create and Configure Azure SQL Server and SQL Database

Create a new Azure SQL server called `blazor-boilerplate-demo-sql` in the `blazor_boilerplate_demo_rg` resource group (very simple to do so instructions not shown here).

Select **Show Firewall Settings** and ensure that **Allow Azure services and resources to access this server** is turned **on**.



Select **+ Create Database** and configure the new database as follows:

Database Name: blazor-boilerplate-demo-db

Elastic Pool: No

Compute/Storage: Basic 5DTU/2GB (about US\$5 per month)

Admin Login: <Your_Admin_Username>

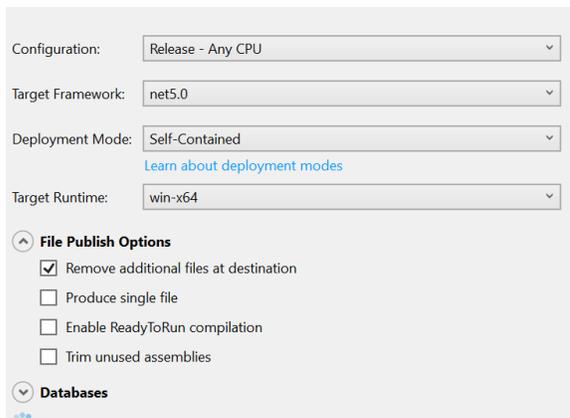
Admin Password: <Your_Admin_Password>

Once created, open the new SQL database resource and select **Show database connection strings**. Copy the **ADO.NET (SQL Authentication)** string and save it for use in the next step.

Configure Visual Studio Project for Azure Publishing

Open the Blazor Boilerplate solution in VS and select the BlazorBoilerplate.Server project. Right-click/Publish and select **+ New** to create a new profile. Select **Import Profile** and now browse and select the Azure Web Deploy publish profile you downloaded from the Overview tab earlier.

Edit the new profile and configure as follows:



Configuration: Debug - Any CPU

Framework: .Net5.0

Deployment Mode: Self-Contained

Runtime: win-x64

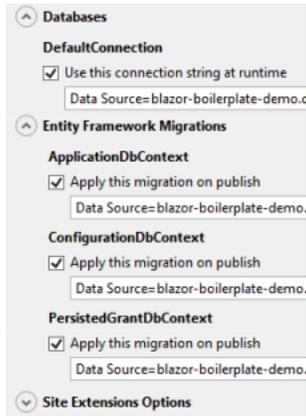
File Publish: Remove additional files at destination

Databases/Default: Use this connection string at runtime

Now paste the SQL database connection string you saved earlier and make this modification. Change 'Server' to 'Data Source'. The final string should look something like this:

```
Data Source=tcp:blazor-boilerplate-demo-sql.database.windows.net,1433; Initial Catalog=blazor-boilerplate-demo-db;User Id=<adminusername>; Password=<adminuserpassword>; Trusted_Connection=False; Encrypt=True; MultipleActiveResultSets=True;
```

Paste the same connection string into each of the **Entity Framework Migrations**.



Select the **Connection** tab and confirm that the **Destination URL** is `http://blazor-boilerplate.azurewebsites.net` or change it to your custom one (`http://blazor-boilerplate.demodomain.com`) if you are using a custom domain. This is the URL that the publish tool will open after publishing.

Create and Configure Azure Key Vault

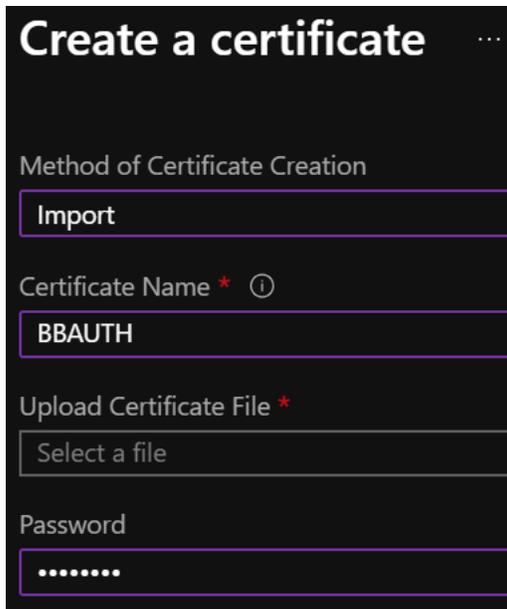
Azure Web App could be running multiple instances in multiple deployment slots. By default, the app stores encrypted information in local storage and separate instances can't access it. We need to store it in a central, protected place that can be accessed by all instances, and all instances must be able to unencrypt the content.

We use Azure Blob Storage as the central store (persistence provider) and Azure Key Vault as the common encryption provider.

Create a new Azure Key Vault in the `blazor_boilerplate_demo_rg` resource group and name it `blazor-boilerplate-demo-kv`.

Open the new resource, select **Certificates** and choose **+ Generate/Import**, then import your .pfx certificate.

Give the certificate the name `BBAUTH` and password `Admin123`.



Create a certificate ...

Method of Certificate Creation

Import

Certificate Name * ⓘ

BBAUTH

Upload Certificate File *

Select a file

Password

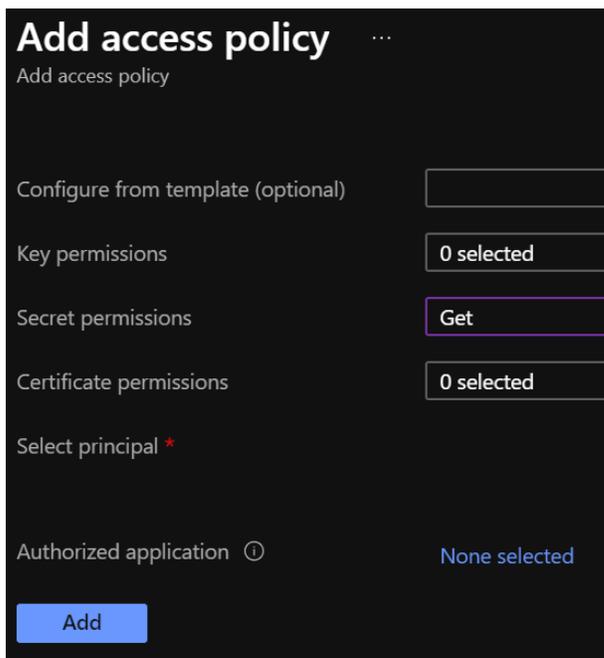
.....

Select the certificate in the list and check that **Issuance Policy / Advanced Policy Configuration** has **Exportable Private Key** set to Yes.

Select and open the certificate and copy the **X.509 SHA-1 Thumbprint (in hex)** for use later in appsettings.json.

Return to the key vault main menu and select **Access Policies**. Add a new policy, then click **Select Principal** and search for / select the Azure App Service you created earlier (e.g. blazor-boilerplate).

Give the policy Get access to **Secret Permissions**, because Identity Server needs access to the private key so we will import the certificate as a secret. It should also have Get access to **Certificate Permissions**.



Add access policy ...

Add access policy

Configure from template (optional)

Key permissions 0 selected

Secret permissions Get

Certificate permissions 0 selected

Select principal *

Authorized application ⓘ None selected

Add

Copy the **DNS Name** of the key vault and save it so that you can add it to appsettings.json in a later step.

Create and Configure Blob Storage (for keys.xml)

In Startup.cs we persist keys to Azure Blob Storage and protect them with Azure Key Vault. We therefore need access to a private blob container to store the keys.

Create an **Azure Storage Account** in the blazor_boilerplate_demo_rg resource group and name it blazor-boilerplate_storage or similar.

Open the new resource and select **+ Container** to create a new blob container called blazor-boilerplate-keys. The access level should be private.

Return to the storage account and select **Access Keys**, then unhide the keys. Copy the key1 or key2 **Connection String** and save it so that you can add it to **appsettings.json** in the next step.

Configure Blazor Boilerplate & Deploy to Azure

Configure Azure settings in appsettings.json

In Visual Studio, open the BlazorBoilerplate solution if it isn't already, then select the **BlazorBoilerplate.Server** project. Edit **appsettings.json**.

Set **DefaultConnection** to the string we stored earlier, e.g:

```
Data Source=tcp:blazor-boilerplate-demo-sql.database.windows.net,1433; Initial Catalog=blazor-boilerplate-demo-db; User Id=<adminusername>; Password=<adminuserpassword>; Trusted_Connection=False; Encrypt=True; MultipleActiveResultSets=True;
```

Edit both of the following sections and insert your own saved parameters where shown:

HostingOnAzure

```
"RunsOnAzure": true,
```

```
"RunningAsAppService": true,
```

```
"RunningAsDocker": false, // not implemented yet.
```

```
"AzureKeyVault": {
```

```
  "UsingKeyVault": true,
```

```
  "UseManagedAppIdentity": true,
```

```
  "AppKey": "", // not implemented yet.
```

```
  "AppSecret": "",
```

```
  "KeyVaultURI": "https://blazor-boilerplate-demo.vault.azure.net/",
```

```
  "CertificateIdentifier": "https://blazor-boilerplate-demo.vault.azure.net/certificates/BBAUTH/<HEX_STRING_HERE>",
```

```
  "CertificateName": "BBAUTH",
```

```
  "ContainerName": "blazor-boilerplate-keys",
```

```
  "KeysBlobName": "keys.xml"
```

```
}
```

BlazorBoilerplate

```
"ApplicationUrl": "https://blazor-boilerplate.demodomain.com",
```

```
"IS4ApplicationUrl": "https://blazor-boilerplate.demodomain.com",
```

```
“UseLocalCertStore”: false,  
“CertificateThumbprint”: “<X.509_SHA-1_THUMBPRINT_HERE>”,  
...
```

You may also want to change **Serilog / MinimumLevel / Default** from ‘Warning’ to ‘Debug’ while you are getting the demo up and running.

Check / Modify Startup.cs

In Visual Studio, open the BlazorBoilerplate solution if it isn’t already, then select the **BlazorBoilerplate.Server** project. Edit **Startup.cs**.

The section that relates to Azure hosting begins around line 147. Find the two lines below:

```
dataProtectionBuilder.PersistKeysToAzureBlobStorage(blobClient);
```

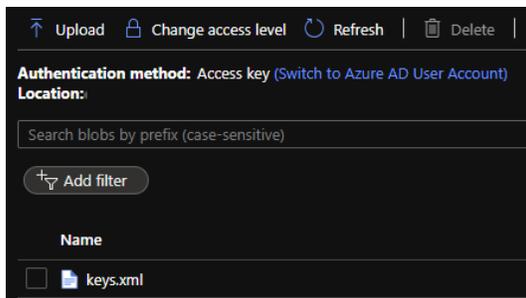
```
dataProtectionBuilder.ProtectKeysWithAzureKeyVault(new Uri(certificateIdentifier), new Default-  
AzureCredential(credentialOptions));
```

There is a current limitation of persisting keys to blob storage. The blob won’t get created on first run, and the app will probably error on startup. To fix this, comment out the line **dataProtectionBuilder.ProtectKeysWithAzureKeyVault**, publish the app and let it run, and verify that **keys.xml** is created in the keys blob container before you uncomment it.

Publish the BlazorBoilerplate Solution

Right-Click the **BlazorBoilerplate.Server** project and select **Publish**. Hit the Publish button and check that the app publishes without errors and opens a browser with the URL you specified earlier in the publish profile.

If the app was published successfully, you can now check that it managed to persist keys to blob storage. Open your **Storage Account** and select **Containers**. Open the blazor-boilerplate-keys container and confirm that **keys.xml** now exists. If so, proceed to the final step below.



The site should now redirect to <https://blazor-boilerplate.demodomain.com> or <https://blazor-boilerplate.azurewebsites.net> if you aren’t using a custom domain, and open correctly at the Blazor Boilerplate home page.

Azure Troubleshooting Tips

Kudu

The **Kudu Diagnostic Console** is available at <https://blazor-boilerplate.scm.azurewebsites.net/DebugConsole>.

A few of the more useful troubleshooting logs are:

/LogFiles/stdout_??_?.log.

/site/wwwroot/Logs/log-???.log. If you set the **SerilogMinimumLevel** to Debug earlier you will see the full series of startup log entries, including any errors related to startup.

You can also stream logs from the web app either within the Azure Portal (Web App Service / Monitoring / Log Stream) or to Visual Studio if you prefer.

Remote Debug in Visual Studio

To remotely debug, you must first publish a **Debug Configuration** of Blazor Boilerplate to Azure. Edit the **Publish Profile** in Visual Studio and set **Configuration** to Debug - Any CPU.

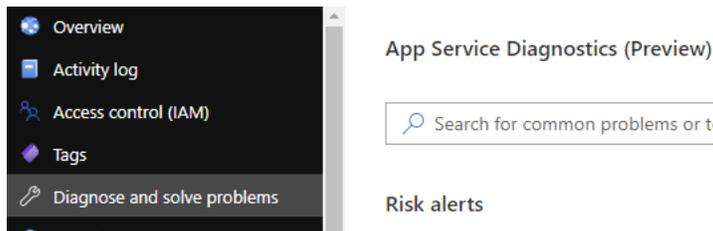
As a simple remote debugging test, try the following:

Open the **Shared / Modules** folder and the **BlazorBoilerplate.Theme.Material.Demo** project. Open the **Pages** folder and edit **ToDoList.razor**. Set a breakpoint at the line **await LoadMainEntities();**

Open **Cloud Explorer** in Visual Studio and select your web app within your subscription (under App Services), right-click and **Attach Debugger** to the Blazor Boilerplate app. The Visual Studio Output Window will show the application starting up. Once symbols are loaded a browser should open and display the home page. Select **ToDo List** and execution should halt at the **await LoadMainEntities();** breakpoint in VS.

Azure Portal Web App Diagnostic Tools

App Service Diagnostics as accessed via **Diagnose and Solve Problems** within the Web App on the Azure Portal.

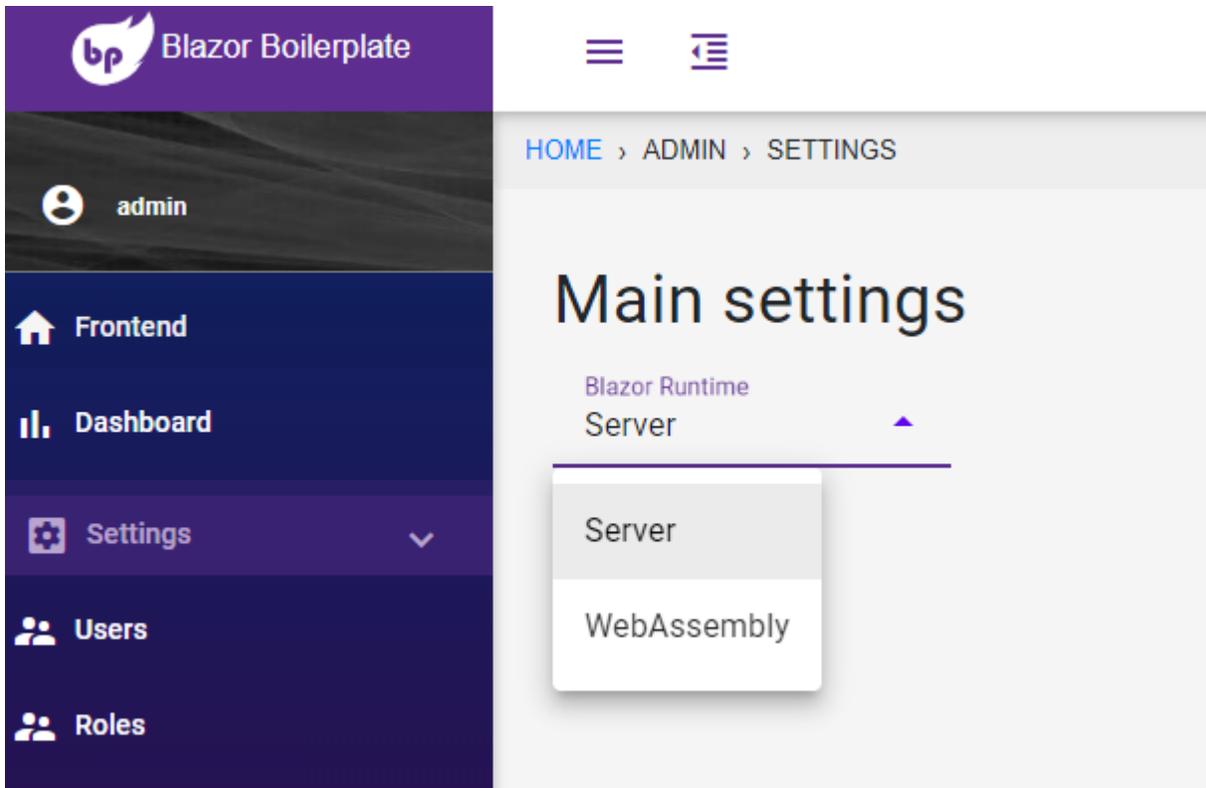


Diagnostic Tools has a couple of useful tools, including **Check Connection String**, access to **Application Event Logs**, and **Advanced Application Restart**

2.2.8 Dual mode Blazor

As stated in *doc home* Blazor has two hosting models: **WebAssembly** or **Server**. Each hosting model has pros and cons, so you have to decide which is better for your web application.

With Blazor Boilerplate you can switch at runtime between WebAssembly or Server in the Settings of Admin.



Choose the desired runtime in the dropdown list and then click on Save button. The application is then reloaded.

2.2.9 MultiTenancy

The implementation of multitenancy is based on `Finbuckle.MultiTenant`.

The host strategy is used:

```
services.AddMultiTenant()
    .WithHostStrategy("__tenant__")
    .WithEFCoreStore<TenantStoreDbContext>()
    .WithFallbackStrategy(Settings.DefaultTenantId);
```

Setup Visual Studio and Windows for multiple bindings

Open `\src.vs\BlazorBoilerplateconfig\applicationhost.config` and add these bindings:

```
<site name="BlazorBoilerplate.Server" id="...">
  <application path="/" applicationPool="BlazorBoilerplate.Server AppPool">
    <virtualDirectory path="/" physicalPath="...\src\Server\BlazorBoilerplate.Server" />
  </application>
  <bindings>
    <binding protocol="http" bindingInformation="*:53414:localhost" />
    <binding protocol="http" bindingInformation="*:53414:tenant1.local" />
    <binding protocol="http" bindingInformation="*:53414:tenant2.local" />
  </bindings>
</site>
```

Run as administrator `\src\Utils\Scripts\addTenantBindings.cmd` to enable access in Windows to the above bindings. It contains commands like

```
netsh http add urlacl url=http://tenant1.local:53414/ user=everyone
```

Open as administrator `C:\Windows\System32\drivers\etc\hosts` and add the following line:

```
127.0.0.1          tenant1.local tenant2.local
```

Delete the previous database if any, because DatabaseInitializer inits TenantInfo for the two tenants.

Run debug and in Admin UI you will find a MultiTenancy section with links to the two demo tenants.

In the following screenshot you see the configuration with the two online demo tenants.

The screenshot shows the Admin UI for Blazor Boilerplate. The left sidebar contains navigation items: Frontend, Dashboard, Settings, Users, Roles, Multi-tenancy (selected), Identity Server, and Monitoring. The main content area is titled 'Tenants' and 'Tenants Management'. It features a table with the following data:

+ NEW TENANT	Id	Identifier	Name
	Master	Master	Master
	tenant1	blazor-server.quareella.net	Microsoft Inc.
	tenant2	blazor-wasm.quareella.net	Contoso Corp.

2.2.10 IdentityServer4

IdentityServer4 has detailed documentation to read first.

IdentityServer4 authentication is used with ApplicationController:

```
[Route("api/data/[action]")]
[Authorize(AuthenticationSchemes = AuthSchemes)]
[BreezeQueryFilter]
public class ApplicationController : Controller
{
    private const string AuthSchemes =
        "Identity.Application" + "," + IdentityServerAuthenticationDefaults.
        AuthenticationScheme; //Cookie + Token authentication
```

In fact as you can see ApplicationController uses both cookie and bearer token authentication scheme.

Currently ApiClient uses cookie authentication to access ApplicationController. To see an example of external access with ApiClient and bearer authentication, you have to look at **BlazorBoilerplate.IdentityServer.Test#** projects.

2.2.11 Source Generators

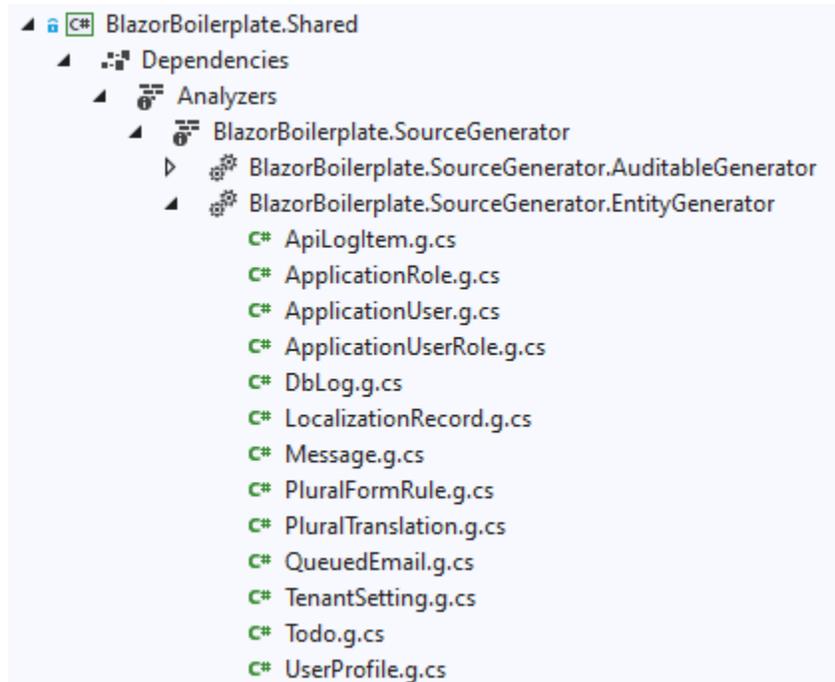
To avoid repetitive tasks, I created some [Generators](#) based on [Source Generators](#).

Notes

Visual Studio 16 has some issues with Source Generators, especially when you clone BlazorBoilerplate and build the first time.

Even rebuilding **BlazorBoilerplate.Shared** sometimes a lot of errors are listed, but they are fake. Ignore them and run **BlazorBoilerplate.Server** or close and reopen Visual Studio.

Source generated is not visually updated.



Sometimes entities generated and present in the figure do not respect the most updated version of the source.