C# Full Stack Deployment

This document contains a checklist of steps to host your Angular/C# application on the cloud with Microsoft Azure.

Overview

After creating a free account on Microsoft Azure, you will need to set up three separate services.

1. Azure SQL Database to store your data.
2. Azure App Service to run your C# Backend API. This will connect to the database.
3. Azure Static Web App to host the Angular web application. This will connect to C# API.

It's important to set up these services in order since the later ones depend on the earlier ones. The steps below will guide you through setting up each service.

Free account notes

Even though this is a free account, you will need to supply a credit card to put on file. This will prevent multiple accounts from being made with the same card.
Once you have a free account setup, you will have the following restrictions on your account.

- A starting $200 credit limit. This will get used before your credit card. Your card should not get charged once you run out until you tell Azure to continue.
- A deployment limit of about 10 deployments. You shouldn't run into this if everything goes smoothly. If you run into any errors, get help before you deploy again.

Initial Project Setup

When you first start your project, set it up as follows.

Database: Each team member will usually use their own local database.

API: Create a C# API backend in Visual Studio. Push this to its own GitHub repository.

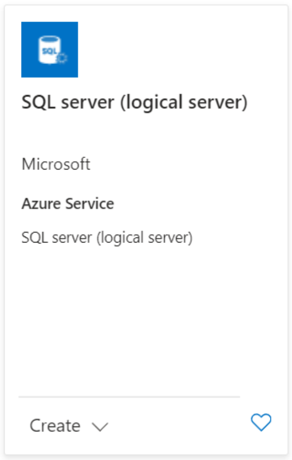
Angular: Use ng create to make an Angular project separate from the C# API project. Use Visual Studio Code as the IDE. Push this to its own separate GitHub repository as well.

Note: The API and Angular will need to be in separate GitHub repositories in order for the later Azure deployment steps to work correctly.

SQL Database

Make sure to carefully complete every step below in order to set up your SQL database in the Azure cloud.

On Azure, click “Create a resource”. This option will either show up on the home page or by opening the hamburger menu. Then search for “SQL server (logical server)”.



Once you click create on it, a form will appear. Here are the inputs on it and what they mean.

- Subscription. This is related to whatever credit card you set up on account creation. Leave this on whatever the default is.
- Resource Group. This is where all the resources you will be created will be grouped. Create a new resource group and name it after your project.
- Server. You need to create a server.
    - Server name. Name it whatever you want, but it must not be taken by anyone else.
    - Location. Set it to the closest location to you.
    - Authentication. Set to SQL authentication
    - Add a login. Write this down and make sure you don’t mind sharing this with your team.

Click Next: Networking >

- Firewall Rules. Set to yes. This will allow you to use this database on the project.

The rest do not matter to us, so review and create. On that page, click create.

Give it 5+ minutes to deploy.

Once it's deployed, click go to resource.

Look for the part called Server name and grab the url it supplies. Copy it and save it somewhere.

Next, to allow everyone on the team to use the database, click overview on the left. On the top of the page, there should be a button called "Set server firewall". It will bring you to a new page. You should find a section called "Firewall rules" and in that is a button called "Add a firewall rule". Click that button.

- For the Rule name, name that after your teammate.
- For both the Start and End IP, grab your IPv4 from this website. https://www.whatismyip.com/
- Create one rule for each teammate.

On this page, make sure the "Allow Azure services and resources to access this server" button is clicked.
Once this is all done, click save on the bottom.

NOTE: To make the free offer last longer, follow the following sections to create a database and lower their power.

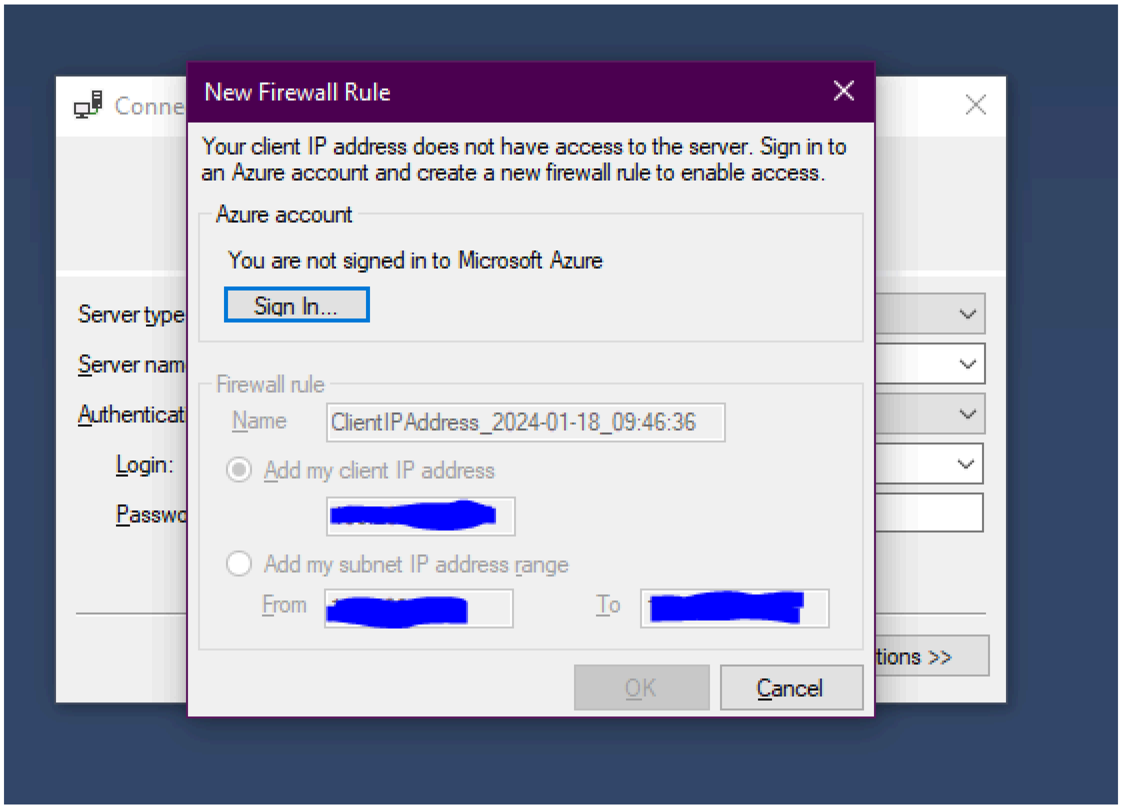
Testing Server

Once you are ready to go, open up SSMS.

- Change the server name to the one you copied from Azure.
- Set the Authentication to SQL Server Authentication
    - Use the Login and Password you created.

Click Connect.

- If it all worked, you should connect to the server with no issue.
- If you see the following window, then click sign in. Even if your Azure account isn't the one that made the Database, you need to sign in. Once you do, the firewall rule section should open up. Change the name to your name and click ok.


New Firewall Rule

Your client IP address does not have access to the server. Sign in to an Azure account and create a new firewall rule to enable access.

Azure account

You are not signed in to Microsoft Azure

Sign In...

Firewall rule

Name: ClientIPAddress_2024-01-18_09:46:36

Add my client IP address

[redacted]

Add my subnet IP address range

From [redacted] To [redacted]

Once you are connected, you can go ahead and make your tables. You'll all be working on the same database. So any changes you make, your teammates will see. When you are ready to use this database with Entity Framework, head down to the C# Backend's section on Entity Framework.

SAVE MONEY (Important)

Azure uses a free credit system and we start with a $200 starting limit. If we don't go over that amount, then we won't lose money. To avoid running out of credit, we will change how powerful our database is.

Once you've created the database, head to your server on Azure. You will see a section below with the new database. Click on that and click "Compute + Storage". On the Service tier dropdown, select "Basic", then set the Data max size to the smallest allowed. (0.1) Once set, hit apply. This will take a few minutes and may prevent you from making changes until it's done.

C# Backend

For our backend, we will set up a project using ASP.Net Core Web API.

Entity Framework

Database first

This is the normal command. We will be adjusting this to work with our Azure Database.

```
Scaffold-DbContext 'Data Source=.\sqlexpress;Initial Catalog=CarDB;
Integrated Security=SSPI;Encrypt=false;TrustServerCertificate=True;'
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

- Remove Integrated Security=SSPI;
- Remove Encrypt=false;
- Remove TrustServerCertificate=True;
- Adjust Data Source by replacing .\sqlexpress with your new Server name you copied earlier.
- Adjust Initial Catalog by replacing CarDB with your database name inside the server.
- Add User Id=YourAzureServerUserName;
- Add Password=YourAzureServerPassword;

Here's an example after making these changes.

```
Scaffold-DbContext 'Data
Source=demoproject123.database.windows.net;Initial Catalog=UserDB;
User Id=GCAdmin; Password=GrantChirpus123'
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Code First

Run through the steps to create Code first. Once you are getting to your db context, we will be adjusting the connection string.

```
'Data Source=.\sqlexpress;Initial Catalog=CarDB; Integrated Security=SSPI;'
```

- Remove Integrated Security=SSPI;
- Adjust Data Source by replacing .\sqlexpress with your new Server name you copied earlier.
- Adjust Initial Catalog by replacing CarDB with your database name inside the server.
- Add User Id=YourAzureServerUserName;
- Add Password=YourAzureServerPassword;

```
'Data Source=demoproject123.database.windows.net;Initial
Catalog=UserDB; User Id=GCAdmin; Password=GrantChirpus123'
```

CORS

CORS (Cross-Origin Resource Sharing) is a system for determining when it is allowed for an application in the browser (such as Angular) to call an API from a particular domain. By default only code running on the same domain and port as the API is allowed to use it..

For example, if Angular is running on localhost:4200, it can only call an API that is also running on localhost:4200. Or if it is running at https://dinosaurlorestore.com, it can only call an API that is also running on https://dinosaurlorestore.com.

When CORS is blocking your API requests, you'll see an error like this:

> Access to XMLHttpRequest at 'http://localhost:8080/orders' from origin 'http://localhost:4200' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

The only way to change this is to configure the API to allow other "origins" (combo of protocol, domain, and port). This must be done in the code for the API. There's nothing you can do on the Angular side to change this.

How to prevent

Add the following lines to the Program.cs. Run your Angular project and replace the localhost port with your development port. If you are hosting, add the website URL to the list as well. (Make sure to come back to this once your Angular App is deployed.

```
//This line should exist
var builder = WebApplication.CreateBuilder(args);
//cors
```

```
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        policy =>
        {
            //replace localhost with yours
            //also add your deployed website
            policy.WithOrigins("http://localhost:4200",

"https://MyChatRoom.com").AllowAnyMethod().AllowAnyHeader();
        });
});
```

Then at the bottom, place this line at the end.

```
//Cors
app.UseCors();
//make sure the cors line is above this
app.Run();
```

Deployment

Visual Studio has a built in way to deploy projects from within. To access it, go to your solution explorer and right click the project name. (Not the solution). Find and click the Publish button.

This should open a new tab with a Publish window. Follow the instructions below to publish correctly.

Note: You may need to login to your Microsoft account. Make sure it's the same one the Azure Project is on. This can be buggy sometimes. If you login, click the x to leave the window and click “Add a publish profile” to re-pull up the menu.

1. Click Azure and then next
2. Click Azure App Service (Windows) and then next.
3. In the middle, click “Create a new instance”
    a. Name whatever you want.
    b. Keep the recommended Subscription name.
    c. Select the Resource Group you made back during the database creation process.

   d. Add a new Hosting plan.
      i. Set the location closest to you
      ii. Set the Size to Free
   e. Click Create. (this may take a few minutes)
4. Click next.
5. In the middle, click “Create a new instance”.
   a. Set API name to match your project.
   b. Keep the recommended Subscription name.
   c. Select the Resource Group you made back during the database creation process.
   d. Add a new API Management Service.
      i. Set the location closest to you.
      ii. The rest should be good.
   e. Click Create. (This may take a few minutes)
6. Click Finish and then close.
7. Once that window is closed, click publish on the top right.
8. After waiting a bit, the API should open on your internet browser. You may need to change the URL to match one of your endpoints to see anything.

Updating the deployed API
This is pretty straightforward. To access it, right click the project in the solution explorer and press publish. This should bring you to a new tab. On the top, just click the publish button again and it will update the published API.

Angular Frontend

Environments
If you are running through this document in order, you should have your backend hosted. We will want the ability to call our development backend when developing and our live backend while live. That’s where environments come into play. They allow you to make sure you call the right backend in the right environment.

```
ng generate environments
```

Inside both we should see the following.

```
export const environment = {};
```

Inside environment.ts we will add the following code.

```
export const environment = {
    production: true,
    apiDomain: "https://MyApp.com/api" // replace this with your deployed
backend API's url
  };
```

Inside environment.development.ts we will add the following code.

```
export const environment = {
    production: false,
    apiDomain: "https://localhost:7135" // replace this with your back API's
localhost url
  };
```

The production boolean tells Angular whether or not this should be used in development or production. True for production, false for development.

The apiDomain is a custom variable that we can use throughout the project. Notice how they are both named the same. That is because it will automatically select the right version for the right stage.

Once we have the environment variables setup, we can use it in our project. We will just have to import the environment from the newly created folder.

```
import { environment } from '../../environments/environment';
```

Then we just have to use it. If we create a baseUrl variable to hold it, we can use it for our API calls. We can access the variable by targeting the environment variable and then accessing the property you want to use.

```
baseUrl: string = environment.apiDomain + "/Users";

constructor(private http: HttpClient) {}

getAllUsers(): Observable<User[]> {
    return this.http.get<User[]>(`${this.baseUrl}`);
};
```

Don't forget to add the HTTPClient Provider to the app.config.ts file.

```
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), provideHttpClient()]
};
```

When you run ng serve, it will use the development version. When you run ng build, it will use the production version. You can also have ng serve use the production version by running

```
ng serve --configuration production
```

Fix Node version

Before we host the project on Azure, we need to make the following change.

Head to your package.json file. We need to add/adjust the engine's section of this json file. When we hook this up to Github/Azure, we want to make sure they are using the right version of Node JS, and by default, it isn't. This will make sure it will.

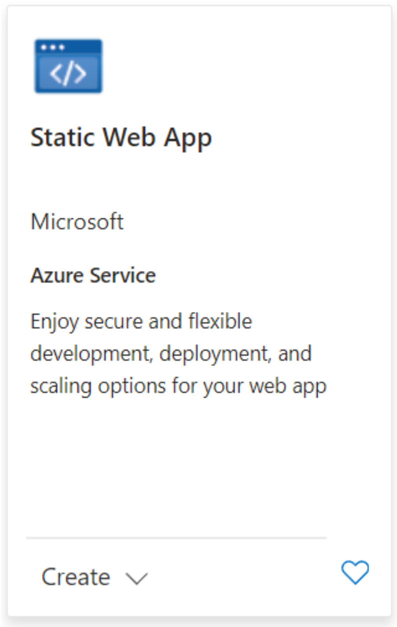
```
"version": "0.0.0",
"engines": {
  "node": ">=20.9.0"
},
"scripts": { ... }
```

Azure Static Web App

Azure will host the built version of the Angular app. The built version is the product of running `ng build` and it consists simply of HTML, JavaScript, CSS, and a few other files that are ready to be loaded directly by the browser.

Using the Static Web App service, Azure will automatically build and deploy the Angular app from our GitHub repository. We just have to make sure everything is set up correctly.

To do this, we will head to Azure and Create a resource. Search for Static Web App.


Static Web App

Microsoft

Azure Service

Enjoy secure and flexible development, deployment, and scaling options for your web app

Click Create on it and you should see a form.

- For the subscription, use the same subscription as the rest of the project.
- For the resource group, set it to the one you've used for the rest of the project.
- For the name, set it to your project name.
- Leave the hosting plan to the free option.
- Set the region closest to you.
- For Deployment details, choose GitHub and login. (Easiest with whoever created the repo)
- Set the organization to your account
- Set the Repository to the one holding your Angular project.
- Set the branch to main (or whatever you want your live branch to be)
- A new input called build presets should show. Choose Angular.
- Leave App location and API location alone.
- On Output location, add dist/ProjectName/browser
    - Replace ProjectName with your project name. Unsure how to write it out? Check out Demoing a build. Or just grab it from the package.json file under "name":"projectname"
- Then hit Review + Create.

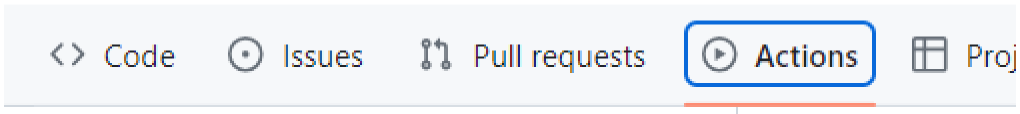
Once it's deployed, click go to resource.

You will see a URL you can follow to see your live project. It might not be 100% ready. Give it some time.

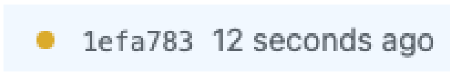
NOTE: This will make a change to your GitHub repo. This will add a file that contains instructions for GitHub to deploy to Azure after every push to the main branch. You will need to do a git pull before you can do more work.

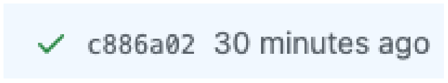
Debugging the deployment

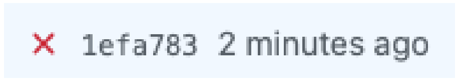
The build and deployment actually happens on GitHub using a feature called GitHub Actions. You will notice an Actions tab on your repository. Go here to see current and past deployments.



The actions run using "workflows". Workflows run automatically when anyone pushes to the repository or merges a pull request. The status is shown near the top of the repository page on GitHub.

- Yellow circle: pending/running 

- Green checkmark: success/tests passed 

- Red "X": failed 

You can click in to see detailed logs if you need to investigate any failures.

NOTE: If there are serious issues trying to deploy using GitHub Actions, ask your instructor about using the Azure Static Web Apps CLI as an alternate approach. (See video.)

Updating the deployed Angular

When you make changes to your Angular app, simply push them to GitHub. After a couple minutes these changes will automatically be deployed to your hosted Azure site. There's nothing else you need to do.

Demoing a build

When you deploy to Azure this way, it will automatically build the project for you. If you deploy it anywhere else, or just want to see what it does, run the following command.

```
ng build
```

This command will build your project into a static web app. You should see a new folder called dist. From there you can go dist/projectname/browser and see all the files it generates.

If you want to deploy to other services, this is most likely the folder you would deploy.

Final edits

Now that everything is deployed, we need to go back and make sure everything is linked to the right endpoints.

Back end

On your Program.cs file, we need to adjust our CORS origins. Make sure the second link is the deployed Angular front end link. Make sure the URL doesn't end with a /. Once this change is made, republish the back end.

```
//cors
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        policy =>
        {
            //replace localhost with yours
            //also add your deployed website
            policy.WithOrigins("http://localhost:4200",

"https://MyChatRoom.com").AllowAnyMethod().AllowAnyHeader();
        });
});
```

Front end

On your environment.ts file, make sure your apiDomain is set to your deployed backend's URL. (If you followed in order, this should be good. Double check though.) Then push the changes to your GitHub.

```
export const environment = {
    production: true,
    apiDomain: "https://MyApp.com/api" //replace this with your deployed
backend API's url
  };
```

Where should I do external API requests?

When you have a full stack project, you might be tempted to do your external API requests on the front end, however there is a major problem with this. If you store API keys on the front end, you cannot hide them. They will be visible in the DevTools by anyone using the site. We also have to store the front end files on GitHub and use that repo to deploy it. If any files are added to the .gitignore on the front end, we won't be able to deploy.

To avoid exposing our API keys, we can do all external API calls in the back end. We can hide the file holding the API keys from GitHub with a .gitignore and the deployed version shouldn't reveal the key.

To do it this way, we will create endpoints on our backend in a controller that calls our external API. Take the response from that API and return it through our controller endpoint. Then on the angular service, we can make calls to that endpoint to get our data.

Pros and Cons of backend external API calls

Pros:	**Cons:**
Can hide API keys from GitHub	Slower call time

Backend is hosted on servers. This will prevent CORS issues	Longer to develop