

Edit Manage Stats



c-arnab

Posted on 16 Jan

Serverless in Azure using Static Web Apps, Functions and Cosmos DB

#azure #azurefunctions #staticwebapps #cosmosdb

In this post, we look at Serverless development on Azure. First we look at the tools and packages necessary to develop locally. Then we create a static site using Azure Static Web Apps (SWA). Then we look at the built in authentication support provided by SWA as we create the authentication layer to ensure users can access their data securely. Then we look at how SWA supports building APIs using built in support for HTTP-triggered functions as we create and Integrate APIs to the static web site using .Net6 and C#. Then we create the data layer in Cosmos DB and integrate the same to functions built earlier. Finally we look at the CI/CD support provided where 'Github repository changes' trigger builds and deploy the solution to Azure.

Problem Statement

The people at helm at the insistence of HR Department have decided to have a Calendar system. Though the solution is supposed to have loads of features, the decision is to build the whole iteratively and a basic Proof Of Concept (POC) is to be developed first. A person at helm attended a conference where s/he heard about "serverless" which allows on-demand scaling and also brings down TCO of a solution with "pay-as-you-go" usage and so one of the requirements for the POC is to make the entire solution using serverless technologies.

Solution

Compute options in serverless services are varied in Azure. One could choose Serverless Containerized Microservices using Azure Container Apps OR Serverless Kubernetes using AKS Virtual Nodes OR Serverless functions using Azure Functions OR the newest entrant- Azure Static Web Apps.

Azure Static Web Apps supports static content hosting, APIs powered by Azure Functions, local development experience, CI/CD workflows, global availability, dynamic scale, preview

environments and all this without the necessity to manage servers, creating & assigning SSL certificates, establishing reverse proxies, etc.

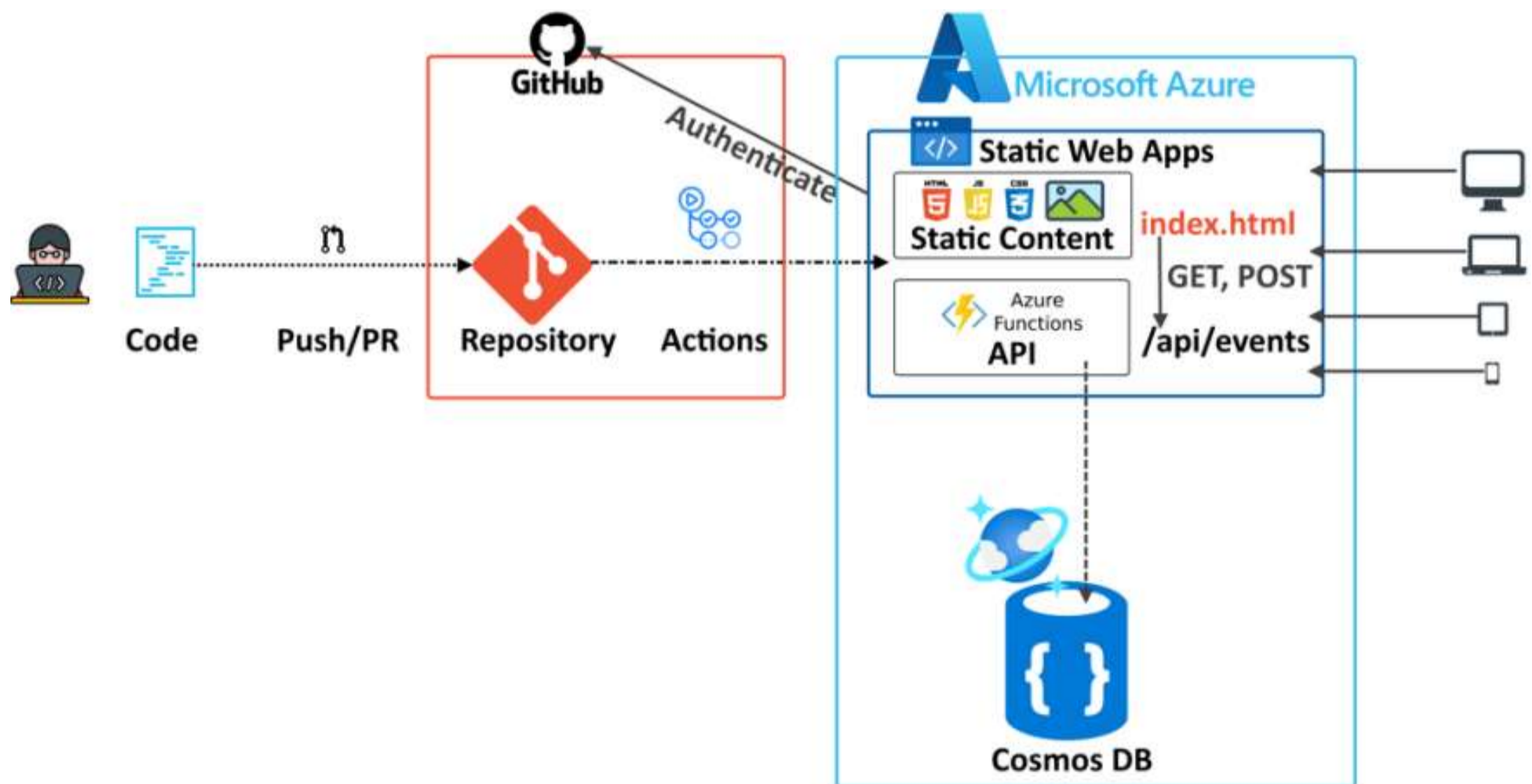
There are two options in Databases amongst serverless services in Azure. The relational Azure SQL Database serverless and the non relational Azure Cosmos DB.

Azure Cosmos DB is a fully managed NoSQL database which offers features such as Change Data Capture and multiple database APIs - NoSQL, MongoDB, Cassandra, Gremlin, & Table enabling one to model real world data using documents, column-family, graph, and key-value data models.

For the POC, the calendar system will enable users to authenticate themselves and view their events as well as add events to the calendar.

Static Web Apps with Cosmos DB will be used to implement these use cases. Visual Studio code will be used as IDE.

Architecture Diagram with Application Development Lifecycle



Prerequisites for local development

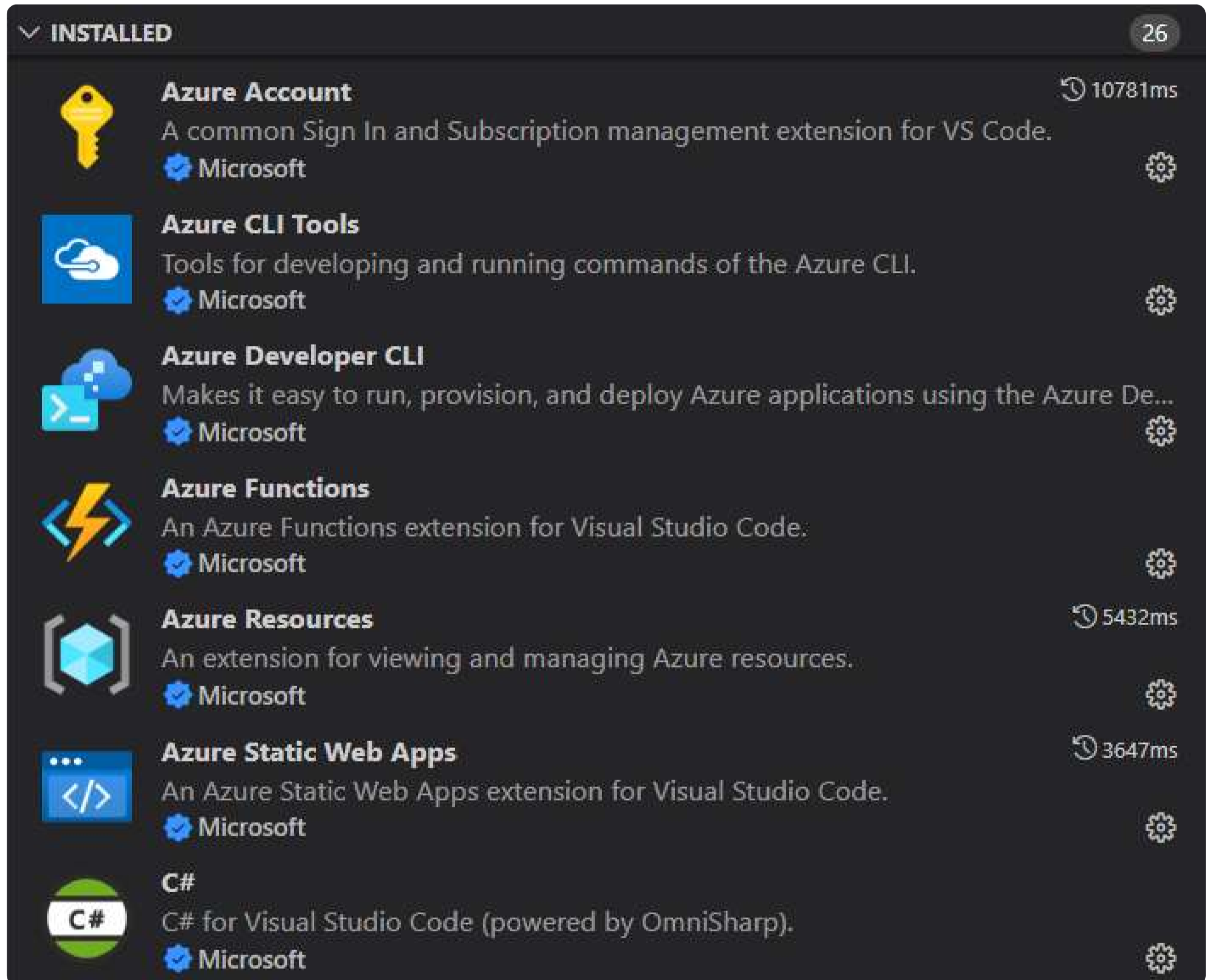
1. .net 6 sdk - <https://dotnet.microsoft.com/en-us/download/dotnet/6.0>
2. Azure Functions Core Tools v4.x - <https://go.microsoft.com/fwlink/?linkid=2174087>
3. Static Web Apps CLI - <https://azure.github.io/static-web-apps-cli/>
4. Azure Cosmos DB Emulator - <https://aka.ms/cosmosdb-emulator>

5. Install Azure Functions Extension from Visual Studio Code Extensions Tab -

<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions>

6. Install Azure Static Web Apps Extension from Visual Studio Code Extensions Tab -

<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurestaticwebapps>



Ensure you have .Net6 sdk (check by running command `dotnet --list-sdks`), even if you have other .Net sdks even a higher one such as .Net7. This is because Azure Functions have concept of In-process and Isolated worker process. This article will have steps supporting In-process whereas .Net7 is only supported in Isolated worker process.

Build Static Site

To get started a template in Github can be used. Go to https://github.com/login?return_to=/staticwebdev/vanilla-basic/generate and in the page add *mycalendar* in the Repository field and click on button *Create Repository from Template* to create repository.

Create a new repository from vanilla-basic

The new repository will start with the same files and folders as [staticwebdev/vanilla-basic](#).

Owner * / Repository name *

Great repository names are short and memorable. Need inspiration? How about [solid-octo-system](#)?

Description (optional)

-  **Public**
Anyone on the internet can see this repository. You choose who can commit.
-  **Private**
You choose who can see and commit to this repository.

- Include all branches**
Copy all branches from [staticwebdev/vanilla-basic](#) and not just `main`.

 You are creating a public repository in your personal account.

[Create repository from template](#)

Plain vanilla javascript template is used here. There are more templates including angular, react, vue, blazor available at <https://github.com/staticwebdev>

Open visual studio code and open a new terminal with bash.

Go to folder where you wish to do your development and run the following command to clone the github project to your local machine.

```
git clone https://github.com/<your_github_account>/mycalendar.git
```

In VSCode select File > Open Folder to open the cloned *mycalendar* repository

Delete files *package.json*, *package-lock.json*, *playwright.config.ts*, entire *tests* folder, entire *.devcontainer* folder and both files (*playwright-onDemand.yml* and *playwright-scheduled.yml*) in *.github/workflows* folder (but do not delete this folder)

Update Github Repository

At the command prompt in terminal, run the following command to update changes in workspace to staging area.

```
git add --all
```

Check the changes to be committed.

```
git status
```

Commit changes from staging to the local repository.

```
git commit -m "Initial commit to create base repository"
```

Push code to Github

```
git push -u origin main
```

```

D:\cosmosdb\mycalendar>git add --all

D:\cosmosdb\mycalendar>git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    .devcontainer/Dockerfile
    deleted:    .devcontainer/devcontainer.json
    deleted:    .devcontainer/library-scripts/node-debian.sh
    deleted:    .github/workflows/playwright-onDemand.yml
    deleted:    .github/workflows/playwright-scheduled.yml
    deleted:    package-lock.json
    deleted:    package.json
    deleted:    playwright.config.ts
    deleted:    tests/Test.README.md
    deleted:    tests/playwright.spec.ts

D:\cosmosdb\mycalendar>git commit -m "Initial commit to create base repository"
[main d163326] Initial commit to create base repository
 10 files changed, 590 deletions(-)
 delete mode 100644 .devcontainer/Dockerfile
 delete mode 100644 .devcontainer/devcontainer.json
 delete mode 100644 .devcontainer/library-scripts/node-debian.sh
 delete mode 100644 .github/workflows/playwright-onDemand.yml
 delete mode 100644 .github/workflows/playwright-scheduled.yml
 delete mode 100644 package-lock.json
 delete mode 100644 package.json
 delete mode 100644 playwright.config.ts
 delete mode 100644 tests/Test.README.md
 delete mode 100644 tests/playwright.spec.ts

D:\cosmosdb\mycalendar>git push -u origin main
Logon failed, use ctrl+c to cancel basic credential prompt.
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 238 bytes | 238.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/c-arnab/mycalendar.git
   e379fb9..d163326  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.

```

Deploy to Azure

In Visual Studio Code, press *F1* OR *Ctrl+Shift+P* to open Command Palette.

Search and Select *Azure Static Web Apps:Create Static Web App...*

In ensuing screens, select your subscription,

select an existing resource group or create a new one (if the screen is shown - not in below image),

decide and select on a free plan or standard plan (if the screen is shown - not in below image),

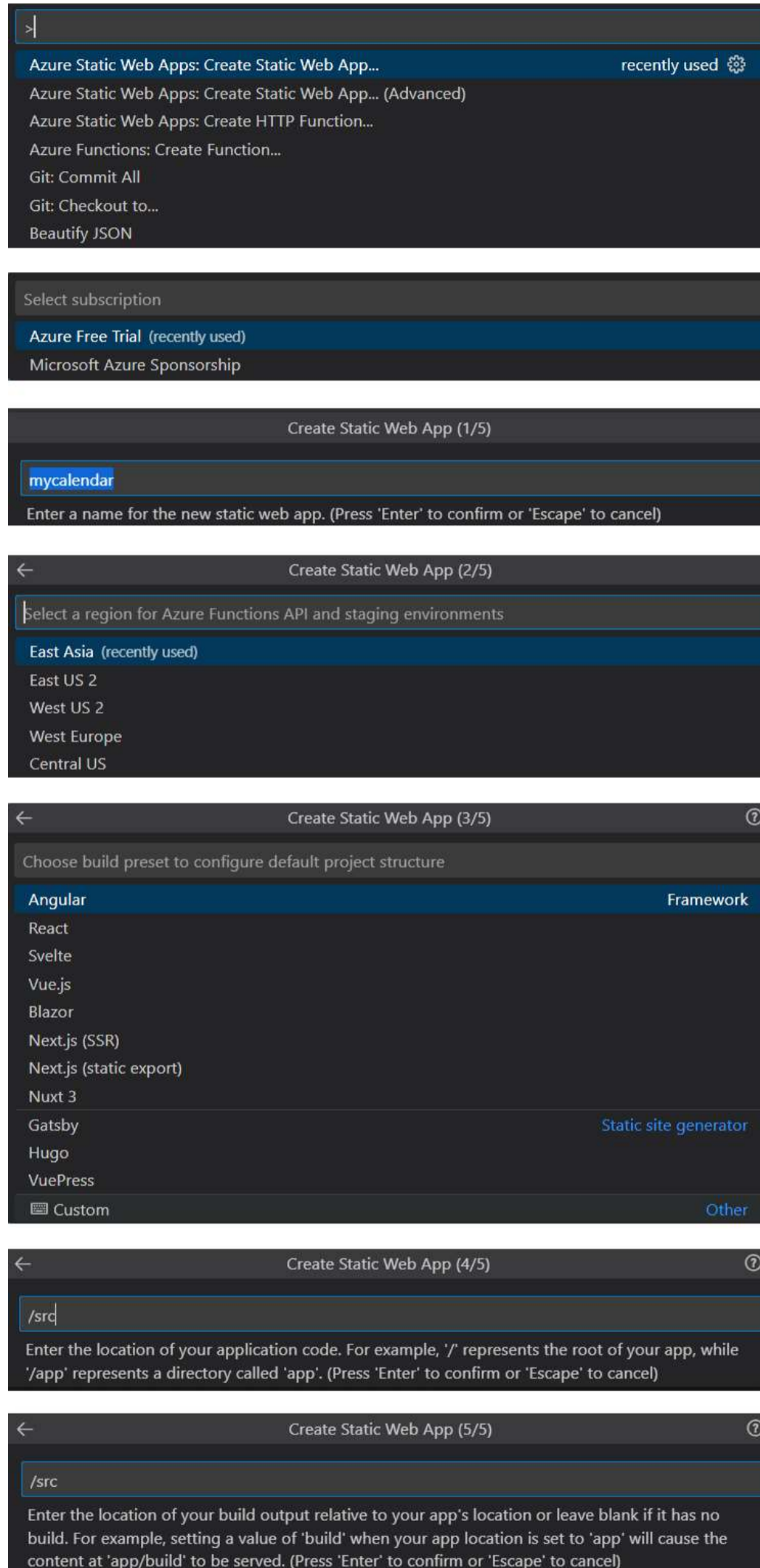
provide a name to the web app,

and then select the region to deploy.

The next screen provides a list of frontend frameworks. As the application is a vanilla

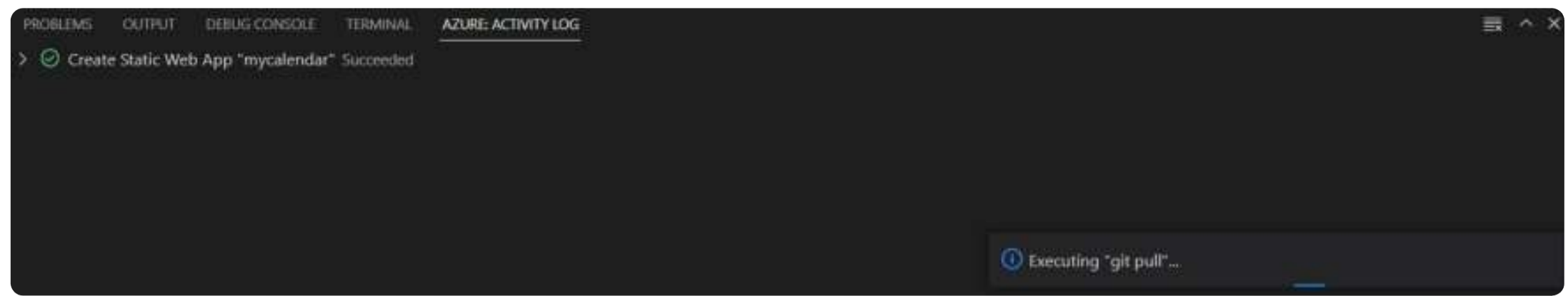
javascript application, choose *Custom*.

Next, provide the location of application code - `/src` as this is where `index.html` resides. Leave the API location blank for now (if the screen is shown - not in below image) and finally provide the location of build output also `/src` (this is primarily useful if a framework such as angular, react, svelte, etc is used and in such cases, the build folder location goes here).



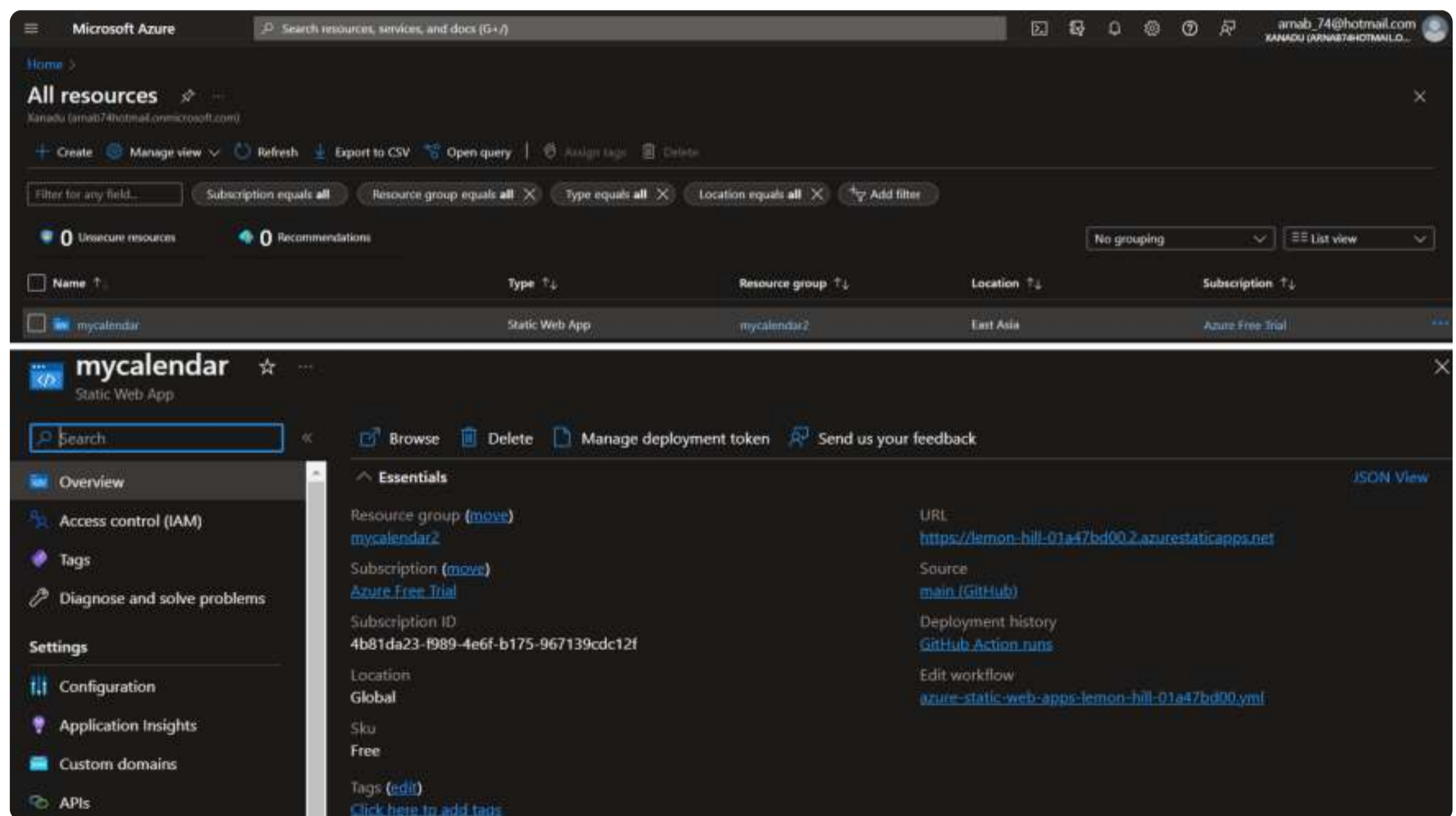
Finally, a message is shown in Azure Activity log stating that the Azure Static Web Site is created and a git pull is executed to download a workflow file from github and kept in `.github/workflows` folder. One of the best things about Static Web Apps is the fact that CI /

CD is integrated using Github Action Workflows and anything that is now added to the workspace and then pushed to github repository will get deployed on Azure automatically.



Azure and Github credentials will have to be provided in the above process.

Check the all resources screen in Azure Portal to find the *mycalendar* static web app. On selecting the same, a page loads which shows the static webapp URL, the source code, action run history and the workflow which was pulled to the workspace (the last three are github links).

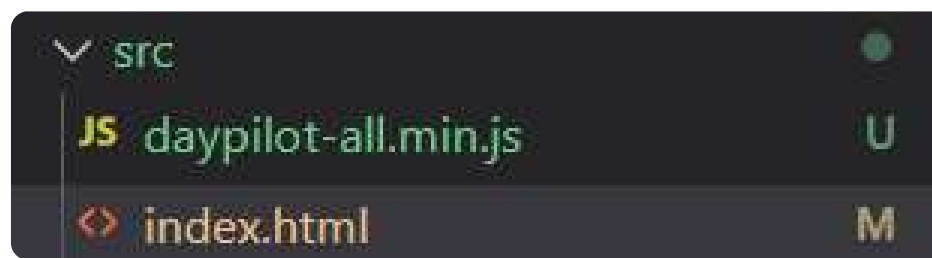


Confirm that the static website URL loads.

Create Application Interface

To create the calendar interface, the free and open-source [DayPilot Lite library](#) is used.

Download the library and add the *daypilot-all.min.js* to the *src* folder.



Update index.html code to the one below.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="daypilot-all.min.js"></script>
  <title>My Calendar</title>
</head>

<body>
  <main>
    <h1>My Calendar</h1>
  </main>

  <div id="mycalendar"></div>

<script type="text/javascript">
  const mycalendar = new DayPilot.Month("mycalendar", {
    startDate: "2023-01-01",
    onTimeRangeSelected: async function (args) {

      const colors = [
        {name: "Blue", id: "#3c78d8"},
        {name: "Green", id: "#6aa84f"},
        {name: "Yellow", id: "#f1c232"},
        {name: "Red", id: "#cc0000"},
      ];

      const form = [
        {name: "Text", id: "text"},
        {name: "Start", id: "start", type: "datetime"},
        {name: "End", id: "end", type: "datetime"},
        {name: "Color", id: "barColor", options: colors}
      ];

      const data = {
        text: "Event",
        start: args.start,
        end: args.end,
        barColor: "#6aa84f"
      }
    }
  });
</script>
</body>
</html>
```

```
    };

    const modal = await DayPilot.Modal.form(form, data);

    mycalendar.clearSelection();

    if (modal.canceled) {
      return;
    }

    mycalendar.events.add({
      start: modal.result.start,
      end: modal.result.end,
      id: DayPilot.guid(),
      text: modal.result.text,
      barColor: modal.result.barColor
    });
  }
});

mycalendar.events.list = [
  {
    "start": "2023-01-12T10:30:00",
    "end": "2023-01-12T15:30:00",
    "id": "225eb40f-5f78-b53b-0447-a885c8e92233",
    "text": "React Interview with Shirish Kumar",
    "barColor": "#cc0000"
  },
  {
    "start": "2023-01-16T12:30:00",
    "end": "2023-01-18T17:00:00",
    "id": "1f67def5-e1dd-57fc-2d39-eb7a5f8e789a",
    "text": "Kubernetes Interview with Ramesh Bhat",
    "barColor": "#3c78d8"
  },
  {
    "start": "2023-01-25T10:30:00",
    "end": "2023-01-25T16:00:00",
    "id": "aba78fd9-09d0-642e-612d-0e7e002c29f5",
    "text": "AAD Interview with Girish C",
    "barColor": "#cc0000"
  }
];

mycalendar.init();
```

</script>

```
</body>
```

```
</html>
```

In the above code, a div with id `mycalendar` is added which is referenced in the javascript. `DayPilot.Month` ensures that the view is month based and the starting date of the calendar is stated using the `startDate` attribute.

`onTimeRangeSelected` section allows the user to add an event by either clicking on a single date or selecting multiple dates by dragging on screen. `DayPilot.Modal.form` provides a form with values to update and Save the event.

`mycalendar.events.list` adds existing event data by adding an array of data in a format the library expects.

In the bash terminal, go to `mycalendar` folder and run the following command.

```
swa start src
```

This command will be available if **Static Web Apps CLI** is installed. **Testing and Debugging** is one of the primary challenges with serverless as application is broken into smaller pieces and replicating the environment locally for developer is hard. **Static Web Apps CLI** solves this problem as we will see later. `src` is the folder where static content including HTML, images, javascript and stylesheets are kept.

On running the command Azure Static Web Apps emulator starts and the calendar application can be accessed at <http://localhost:4280>.

Confirm that events added in code can be viewed as well as new events can be added by clicking on a date.

Implement Authentication

Azure Static Web Apps has support for *GitHub*, *Twitter*, and *Azure Active Directory* for authentication by default. Moreover, Static Web Apps CLI provides authentication emulator to mock responses from the three providers mentioned.

To enable login using *github* update the HTML main content area (between tags) to the content below.

```
<main>
  <h1>My Calendar</h1>
  <p>
    <div id="login" style="display: flex; justify-content: end;"><a href="/.auth/login/git
</main>
```

After authentication, to get access to user information such as user id / email, Azure Static Web Apps provides an API endpoint which means not only do developers not have to implement and maintain any oauth related code but also the endpoint does not face serverless architecture challenges like cold start delays.

Update the `<script>` area with code below under `mycalendar.init()`

```
mycalendar.init(); //Add the code below

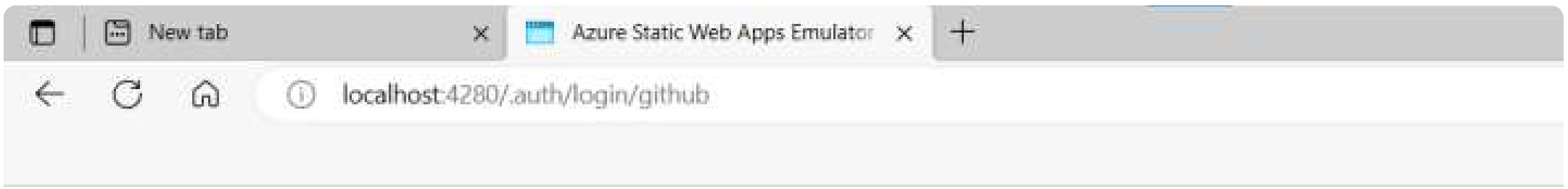
const app = {
  getUserInfo() {
    return fetch('/.auth/me')
      .then(response =>{
        return response.json();
      }).then(data =>{
        const { clientPrincipal } = data;
        console.log(clientPrincipal);
        if (clientPrincipal !=null){
          const userDetails= clientPrincipal.userDetails;
          return userDetails;
        }
        return null;
      })
  },
  init(){
    app.getUserInfo()
      .then(user =>{
        console.log(user);
      })
  }
};
app.init();
</script>
```

In the above code `init` function calls `getUserInfo` function which in turn calls the *direct-access endpoint* `/.auth/me` and from the resultant response gets the *github userid* (provided authenticated by github, else returns null) which gets logged in the browser console.

In the bash terminal, run the following command again.

```
swa start src
```

Load the web page in a browser in incognito mode. After clicking on the login button, the emulator mock screen comes up.



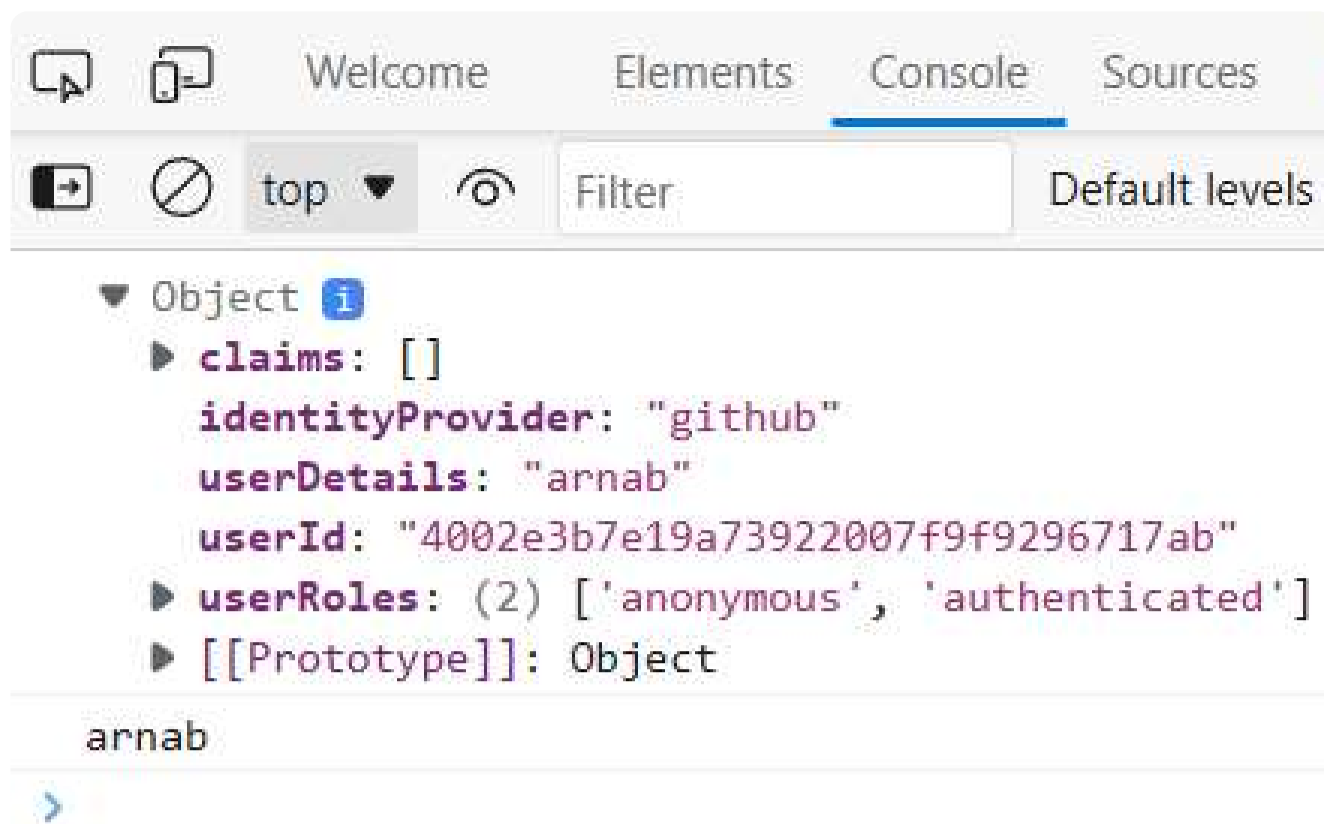
Azure Static Web Apps Auth



Provider	<input type="text" value="github"/> <small>Name of the identity provider</small>
User ID	<input type="text" value="4002e3b7e19a73922007f9f9296717ab"/> <small>An Azure Static Web Apps-specific unique ID for the user</small>
Username	<input type="text" value="arnab"/> <small>Username or email address of the user</small>
User's roles	<input type="text" value="anonymous
authenticated"/> <small>Roles used during authorization. One role per line. Note: roles "authenticated" and "anonymous" will be added automatically if not provided.</small>
User's claims	<input type="text" value="[]"/> <small>Claims from the identity provider. JSON array of claims. See documentation for example claims.</small>
	<input type="button" value="Login"/> <input type="button" value="Clear"/>

In the mock screen, add your first name in the Username field as shown in image (*arnab* is shown in image) and select Login.

The calendar interface shows up as Index.html is loaded. Open developer tools and go to Console. ClientPrincipal data as well as Username is shown as in the image below.



```
Object
  claims: []
  identityProvider: "github"
  userDetails: "arnab"
  userId: "4002e3b7e19a73922007f9f9296717ab"
  userRoles: (2) ["anonymous", "authenticated"]
  [[Prototype]]: Object

arnab
```

Create API and integrate with Static Site

Running logic on the browser has certain limitations namely the ability to connect to data stores / databases to persist data and then retrieve the same. That is where the necessity to run some part of the code server side comes in.

Azure Static Web Apps supports serverless API endpoints powered by Azure Functions where HTTP request triggers the function. The API route is fixed at `/api`. Also, Azure Static Web Apps extension for Visual Studio Code creates the Function templates in `api` folder by default. Also, in local development environment API will run in port 7071 and not port 4280 where the static site runs. This in normal cases will lead to Cross-Origin Resource Sharing (CORS) errors like *Access to XMLHttpRequest at "<http://localhost:7071/api/events>" from origin "<http://localhost:4280>" has been blocked by CORS policy*. But, Azure Static Web Apps (using Reverse Proxy) as well as the CLI (for local development scenario) takes care of this challenge as it makes the static web app and API appear to come from the same domain.

The calendar application allows users to view their existing events as well as add new events to the calendar.

For viewing existing events use case, *GET* method at route endpoint `events` will be used which means that the full api endpoint at the static site will be `api/events`

For adding new events use case, *POST* method at the same endpoint can be used.

View existing events use case

In Visual Studio Code, press `F1` OR `Ctrl+Shift+P` to open Command Palette. Search and Select *Azure Static Web Apps: Create HTTP Function*.

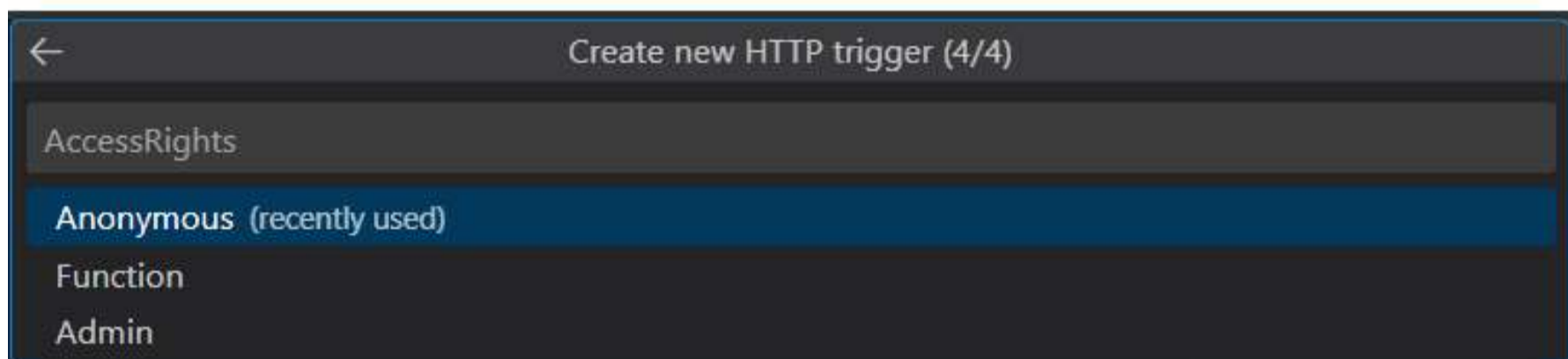
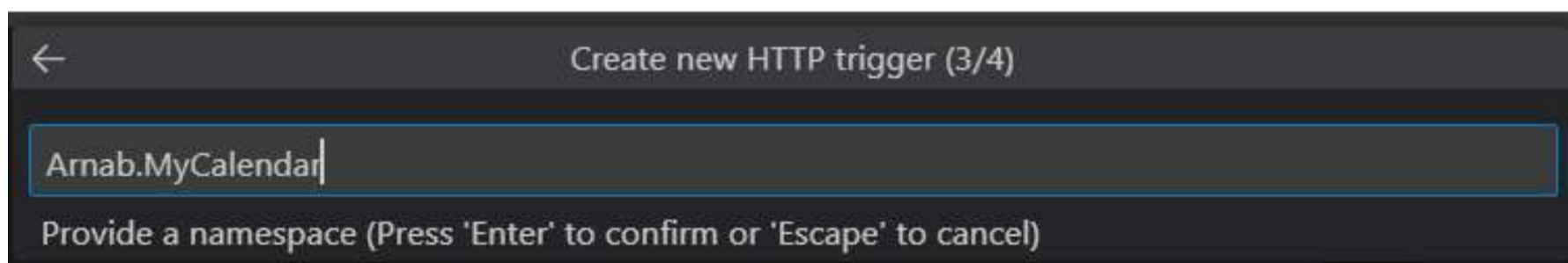
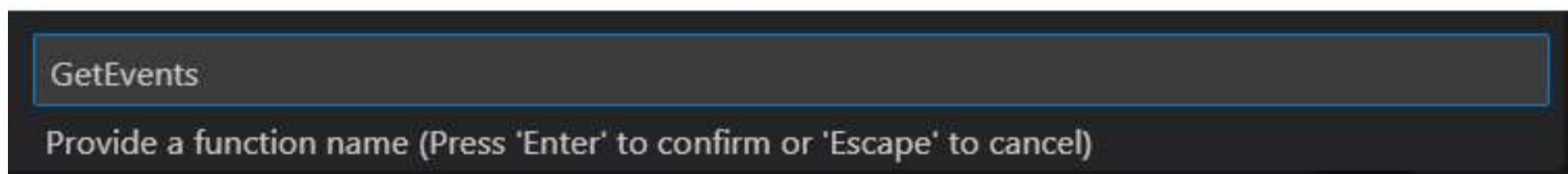
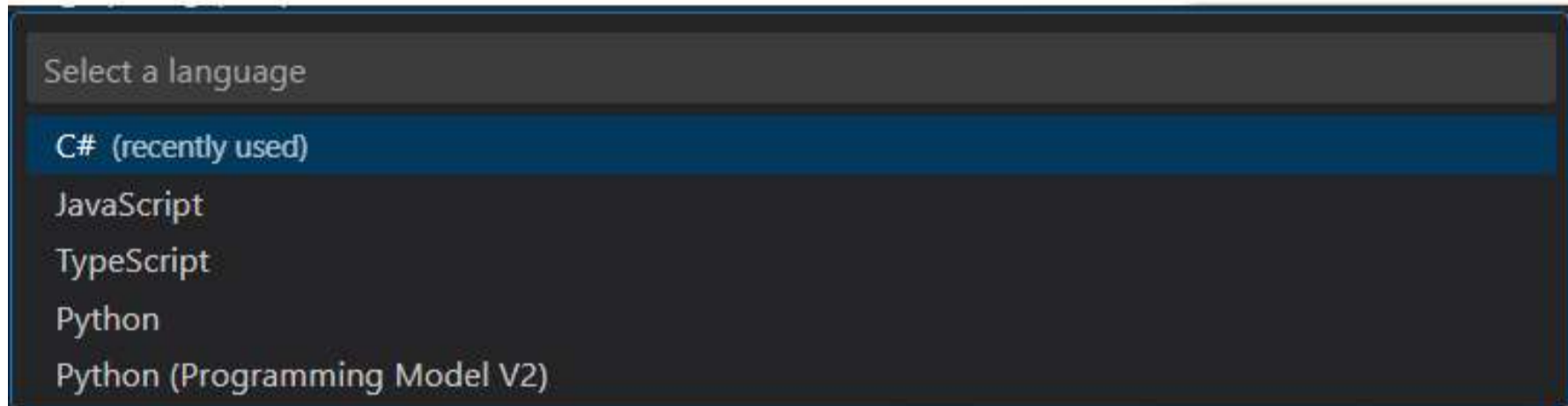
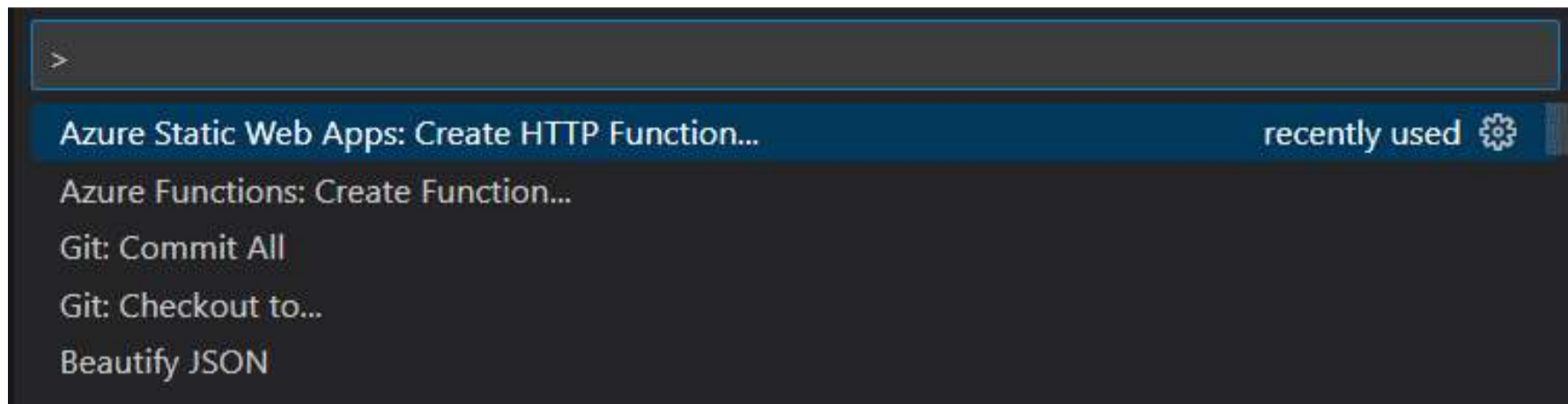
In ensuing screens select `C#` as language,

add `GetEvents` as Function Name,

add `<Your_First_Name>.MyCalendar` as Namespace (*Arnab.MyCalendar* in my case)

and *Anonymous* as Access Rights (Good enough for POC scenarios but never use this setting in production).

Security is important. Do check out [Azure Architecture - Serverless Functions security](#)



A new folder in the workspace gets created named *api* and a Functions project gets created.

A simple piece of code is written next just sufficient to test the integration of API with static site.

In *GetEvents.cs* file update the contents to the code below.

```
using System;
using System.IO;
using System.Threading.Tasks;
```

```

using System.Collections.Generic;
using System.Text.Json;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace Arnab.MyCalendar
{
    public static class GetEvents
    {
        [FunctionName("GetEvents")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "events")] HttpRequest req,
            ILogger log)
        {
            log.LogInformation("Get Events list");
            string username = req.Query["u"];
            log.LogInformation(name);
            string json = @"[
                {
                    ""start"": ""2023-01-12T10:30:00"",
                    ""end"": ""2023-01-12T15:30:00"",
                    ""id"": ""225eb40f-5f78-b53b-0447-a885c8e92233"",
                    ""text"": ""React Interview with Shirish Kumar"",
                    ""barColor"": ""#cc0000""
                },
                {
                    ""start"": ""2023-01-16T12:30:00"",
                    ""end"": ""2023-01-18T17:00:00"",
                    ""id"": ""1f67def5-e1dd-57fc-2d39-eb7a5f8e789a"",
                    ""text"": ""Kubernetes Interview with Ramesh Bhat"",
                    ""barColor"": ""#3c78d8""
                },
                {
                    ""start"": ""2023-01-25T10:30:00"",
                    ""end"": ""2023-01-25T16:00:00"",
                    ""id"": ""aba78fd9-09d0-642e-612d-0e7e002c29f5"",
                    ""text"": ""AAD Interview with Girish C"",
                    ""barColor"": ""#cc0000""
                }
            ]";
            List<Dictionary<string, string>> results=null;
            if (username == "arnab"){
                results =JsonSerializer.Deserialize<List<Dictionary<string, string>>>(json);
            }
        }
    }
}

```



```
        return new OkObjectResult(results);
    }
}
}
```

In the above code we first specify the HTTP method (GET) and route (events).

The API accepts a querystring 'u' which contains the username value.

There is a hardcoded json string on similar lines as in index.html which is deserialized and returned provided the username is equal to a hardcoded value.

The hardcoded value here is *arnab* but you should update that to your first name provided you are going to use that as username in mock authentication screen. Also do remember to change the namespace to `<Your_First_Name>.MyCalendar`.

To call this API from frontend, update *index.html*. Block comment or remove

`mycalendar.events.list` code, add a `loadEvents` function which will call the API and call this function from `init` as code shown below.

```
/* mycalendar.events.list = [
  {
    "start": "2023-01-12T10:30:00",
    "end": "2023-01-12T15:30:00",
    "id": "225eb40f-5f78-b53b-0447-a885c8e92233",
    "text": "React Training",
    "barColor": "#cc0000"
  },
  //more data below
]; */

mycalendar.init();

const app = {
  loadEvents(user) {
    console.log(user);
    var url = new URL('/api/events')
    var params = {u:user}
    url.search = new URLSearchParams(params).toString();
    //console.log(url);
    fetch(url)
      .then(response =>{
        return response.json();
      }).then(data =>{
        //console.log(data);
        mycalendar.update({
          events: data
```

```

        });
    })
}, //next there will be getUserInfo()
init(){
    app.getUserInfo()
        .then(user =>{
            console.log(user);
            if (user !=null){
                app.loadEvents(user);
            }
        })
    }
};
app.init();

```

In the code above `getUserInfo` method is called and if username is not null, `loadEvents` method is called where the username is added to the API endpoint as querystring / search parameters and the return data is updated in the calendar.

To test the API and its integration with the static site, in the bash terminal, run the following command.

```
swa start src --api-location api
```

When the web site loads, login with as username in the ensuing emulator mock screen. The calendar gets updated with data but this time the data is sent from server.

In case you face any challenge / errors in running the above SWA CLI command, open a new bash terminal, ensure you are in api folder, update API urls in index.html from `/api/events` to `http://localhost:7071/api/events` and run the following command to run just the Azure function and in the first terminal run `swa start src` as before. This style of running is also useful in debugging as you bifurcate the running of frontend site and backend API. But, do remember to switch back the urls to its earlier form before publishing the application to Azure.

```
func start --cors http://localhost:4280 --port 7071
```

Add new events use case

Create a new HTTP Function as before and name this PostEvents.

In PostEvents.cs file update the contents to the code below.

```

using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;

```

```

using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

namespace Arnab.MyCalendar
{
    public static class PostEvents
    {
        [FunctionName("PostEvents")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = "events")] HttpRequest req,
            ILogger log)
        {
            log.LogInformation("Post Event");

            string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
            dynamic data = JsonConvert.DeserializeObject(requestBody);
            string cuser = data?.cuser;
            dynamic cevent=data?.cevent;
            log.LogInformation(cuser);
            string eguid =Guid.NewGuid().ToString();
            cevent.id=eguid;
            return new OkObjectResult(cevent);
        }
    }
}

```

In the above code, username and event data is being retrieved from request body, a Guid is created and added to the event data and the event data with Guid is returned.

To call this API from frontend, update *index.html*. Add an `addEvents` function and update the mycalendar `onTimeRangeSelected` function. The final script section in *index.html* is as code shown below

```

<script type="text/javascript">
    const mycalendar = new DayPilot.Month("mycalendar", {
        startDate: "2023-01-01",
        onTimeRangeSelected: async function (args) {

            const colors = [
                {name: "Blue", id: "#3c78d8"},
                {name: "Green", id: "#6aa84f"},
                {name: "Yellow", id: "#f1c232"},
                {name: "Red", id: "#cc0000"},
            ];

```

```

const form = [
  {name: "Text", id: "text"},
  {name: "Start", id: "start", type: "datetime"},
  {name: "End", id: "end", type: "datetime"},
  {name: "Color", id: "barColor", options: colors}
];

const data = {
  text: "Event",
  start: args.start,
  end: args.end,
  barColor: "#6aa84f"
};

const modal = await DayPilot.Modal.form(form, data);

mycalendar.clearSelection();

if (modal.canceled) {
  return;
}

app.getUserInfo()
.then(user =>{
  console.log(user);
  if (user !=null){
    const event = {
      start: modal.result.start,
      end: modal.result.end,
      text: modal.result.text,
      barColor: modal.result.barColor
    };
    app.addEvents(user, event);
  }
})
});

```

```
mycalendar.init();
```

```

const app = {
  loadEvents(user) {
    console.log(user);
    var url = new URL('/api/events')
    var params = {u:user}
    url.search = new URLSearchParams(params).toString();
    //console.log(url);
  }
};

```

```

fetch(url)
  .then(response =>{
    return response.json();
  }).then(data =>{
    //console.log(data);
    mycalendar.update({
      events: data
    });
  })
},
addEvents(user,event){
  fetch('/api/events', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      cuser:user,
      cevent:event,
    }),
  })
  .then(response =>{
    console.log(response);
    return response.json();
  }).then(data =>{
    console.log(data);
    mycalendar.events.add(data);
  })
},
getUserInfo() {
return fetch('/.auth/me')
  .then(response =>{
    return response.json();
  }).then(data =>{
    const { clientPrincipal } = data;
    console.log(clientPrincipal);
    if (clientPrincipal !=null){
      console.log("inside clientprincipal not null");
      const userDetails= clientPrincipal.userDetails;
      return userDetails;
    }
    return null;
  })
},
init(){
  app.getUserInfo()
    .then(user =>{

```

```

    console.log(user);
    if (user !=null){
        document.getElementById("login").style.display = "none";
        app.loadEvents(user);
    }
})
}
};
app.init();
</script>

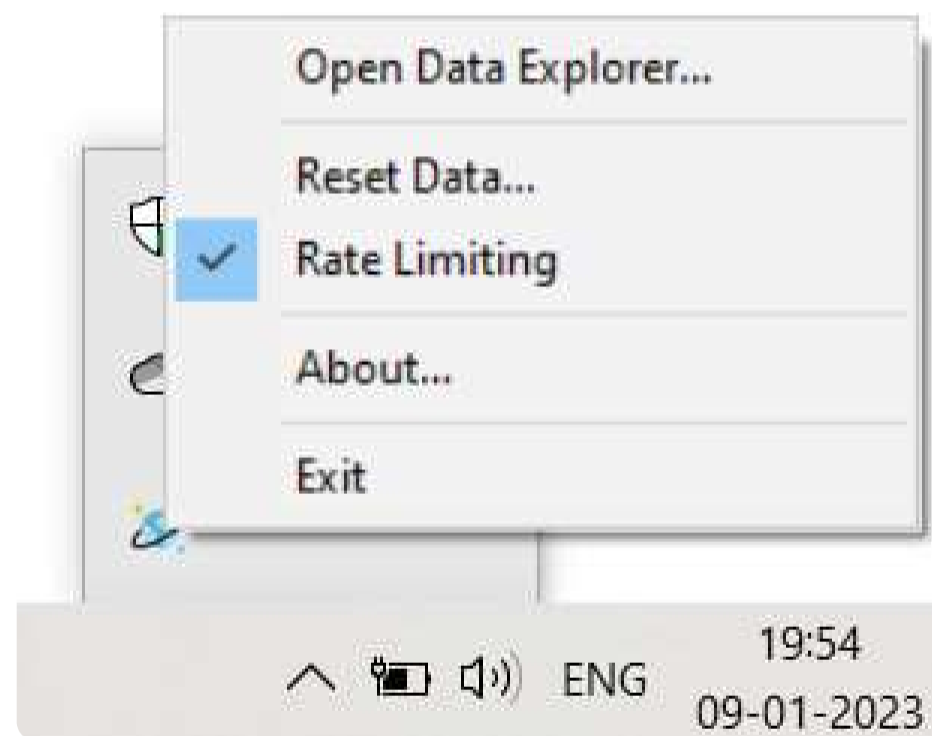
```

The new code in `onTimeRangeSelected` calls `getUserInfo` and if username is not null sends the username and event data to `addEvents` function which makes the POST call to the API and updates the calendar with return data.

Once again test the API and its integration as before to confirm that the code works.

Implement Persistence layer with Cosmos DB

Run the Cosmos DB emulator. Right click and select *Open Data Explorer*.



Run the following command in the bash terminal to configure the connection string (available from *Primary Connection String* in Explorer screen) in the function project settings in the *local.settings.json* file.

```
func settings add CosmosDBConnection "AccountEndpoint=https://localhost:8081/;AccountKey=C
```

local.settings.json file should now be added to the connectionstring as below.

```

"ConnectionStrings": {
  "CosmosDBConnection": {

```

```
"ConnectionString": "AccountEndpoint=https://localhost:8081/;AccountKey=C2y6yDjf5/R+  
"ProviderName": "System.Data.SqlClient"  
}  
}
```

Emulator connection string is same always unless changed manually. Also *local.settings.json* is not pushed to github and the connection string there is only useful for local development. The command / process to add connection string in Azure is different and we will see so later.

In the emulator screen select *Explorer* and then *New Container*.

Add a new Database with name *myCalendar*, state the Container Name as *eventsCollection* and Partition Key */userName*.

New Container ✕

*** Database id** ⓘ

Create new Use existing

Share throughput across containers ⓘ

*** Database throughput (autoscale)** ⓘ

Autoscale Manual

Estimate your required RU/s with [capacity calculator](#).

Database Max RU/s ⓘ

*

Your database throughput will automatically scale from **400 RU/s (10% of max RU/s) - 4000 RU/s** based on usage.

Estimated monthly cost (USD) ⓘ: **\$35.04 - \$350.40** (1 region, 400 - 4000 RU/s, \$0.00012/RU)

*** Container id** ⓘ

*** Partition key** ⓘ

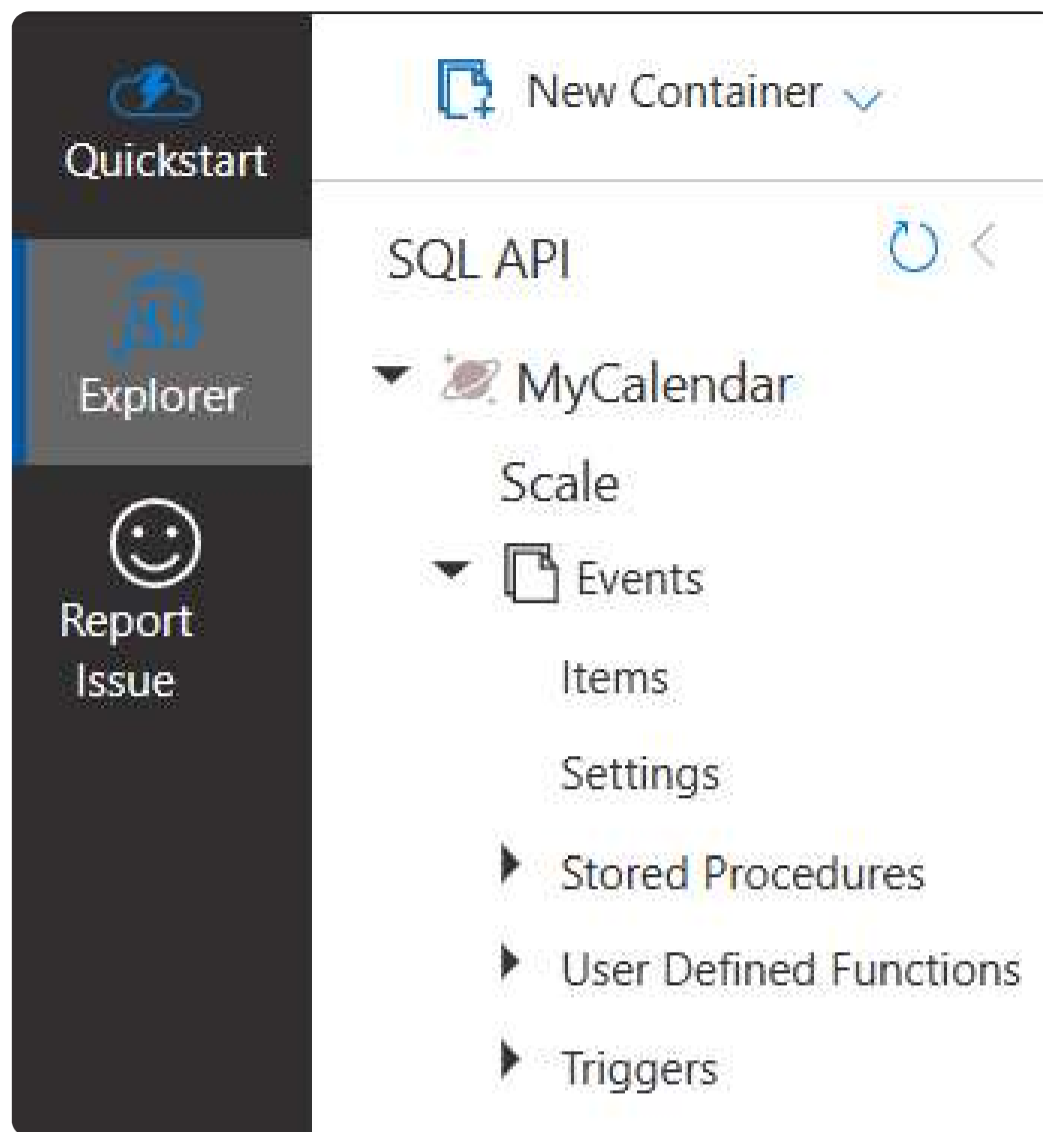
Unique keys ⓘ

+ Add unique key

> Advanced

OK

The database and Container can be viewed now in Explorer.



An easy declarative way to connect Azure services including Cosmos DB to Azure functions is using bindings. Bindings are implemented in extension packages. Run the following dotnet add package command in the terminal to install the Cosmos DB extension package.

```
dotnet add package Microsoft.Azure.WebJobs.Extensions.CosmosDB --version 4.0.0
```

Add a new file in *api* folder and name the same *event.cs*. This file will have two classes, one in the data structure format in which data arrives from frontend and the other is a structure format as the JSON data that will be persisted in Cosmos DB.

```
using System;
namespace Arnab.MyCalendar
{
    public class CalendarEvent
    {
        public string id { get; set; }
        public string userName { get; set; }
        public string startsAt { get; set; }
        public string endsAt { get; set; }
        #nullable enable
        public string? eventTitle { get; set; }
        public string? barColor { get; set; }
        public DateTime eventCreateDate { get; set; }
    }
}
```



```

public class ClientPostEvent
{
    #nullable enable
    public string? id { get; set; }
    public string? start { get; set; }
    public string? end { get; set; }
    public string? text { get; set; }
    public string? barColor { get; set; }
}

public class ClientData
{
    public string cuser { get; set; }
    public ClientPostEvent cevent { get; set; }
}
}

```

In the code above `CalendarEvent` will be used to persist data in Cosmos DB.

Update the contents of PostEvents.cs to the code below.

```

using System;
using System.IO;
using System.Text.Json;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace Arnab.MyCalendar
{
    public static class PostEvents
    {
        [FunctionName("PostEvents")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = "events")] HttpRequest req,
            [CosmosDB(
                databaseName: "myCalendar",
                containerName: "eventsCollection",
                Connection = "CosmosDBConnection")]
            IAsyncCollector<CalendarEvent> eventsOut, ILogger log)
        {
            log.LogInformation("Post Event");

            string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
            ClientData data = JsonSerializer.Deserialize<ClientData>(requestBody);

```

```

string cuser = data?.cuser;
ClientPostEvent cevent=data?.cevent;
log.LogInformation(cuser);
string eguid =Guid.NewGuid().ToString();
cevent.id=eguid;
CalendarEvent nevent = new CalendarEvent() {
    id = cevent.id,
    userName= cuser,
    startsAt=cevent.start,
    endsAt=cevent.end,
    eventTitle=cevent.text,
    barColor=cevent.barColor,
    eventCreateDate=DateTime.Now
};

await eventsOut.AddAsync(nevent);
return new OkObjectResult(cevent);
}
}
}

```

In the code above, the presence of bindings enable in connecting to database seamlessly with the attribute `CosmosDB`. Another parameter `eventsOut` is of type `IAsyncCollector<CalendarEvent>` ensuring that any instance of it calling `AddAsync` gets an instance of `CalendarEvent` persisted in the database.

Update the contents of GetEvents.cs to the code below.

```

using System;
using System.IO;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.Cosmos;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace Arnab.MyCalendar
{
    public static class GetEvents
    {
        [FunctionName("GetEvents")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "events")] HttpRequest req,
            ILogger log, ExecutionContext ctx)
        {
            // ...
        }
    }
}

```

```

[CosmosDB(
    databaseName: "myCalendar",
    containerName: "eventsCollection",
    Connection = "CosmosDBConnection")] CosmosClient client,
ILogger log)
{
    log.LogInformation("Get Events list");
    string name = req.Query["u"];

    List<Dictionary<string, string>> results=new List<Dictionary<string, string>>();
    Container myContainer = client.GetDatabase("myCalendar").GetContainer("eventsCollection");
    QueryDefinition queryDefinition = new QueryDefinition(
        "SELECT * FROM items i WHERE (i.userName = @searchterm)"
        .WithParameter("@searchterm", name);
    string continuationToken = null;
    do
    {
        FeedIterator<CalendarEvent> feedIterator =
            myContainer.GetItemQueryIterator<CalendarEvent>(
                queryDefinition,
                continuationToken: continuationToken);

        while (feedIterator.HasMoreResults)
        {
            FeedResponse<CalendarEvent> feedResponse = await feedIterator.ReadNextAsync();
            continuationToken = feedResponse.ContinuationToken;
            foreach (CalendarEvent item in feedResponse)
            {
                results.Add(new Dictionary<string, string>(){
                    {"start", item.startsAt},
                    {"end", item.endsAt},
                    {"id", item.id},
                    {"text", item.eventTitle},
                    {"barColor", item.barColor}
                });
            }
        }
    } while (continuationToken != null);

    return new OkObjectResult(results);
}
}
}

```

In the above code, a Azure Cosmos DB binding provided `CosmosClient` instance, available in extension version 4.x, is used to read a list of documents. With the access of `CosmosClient`

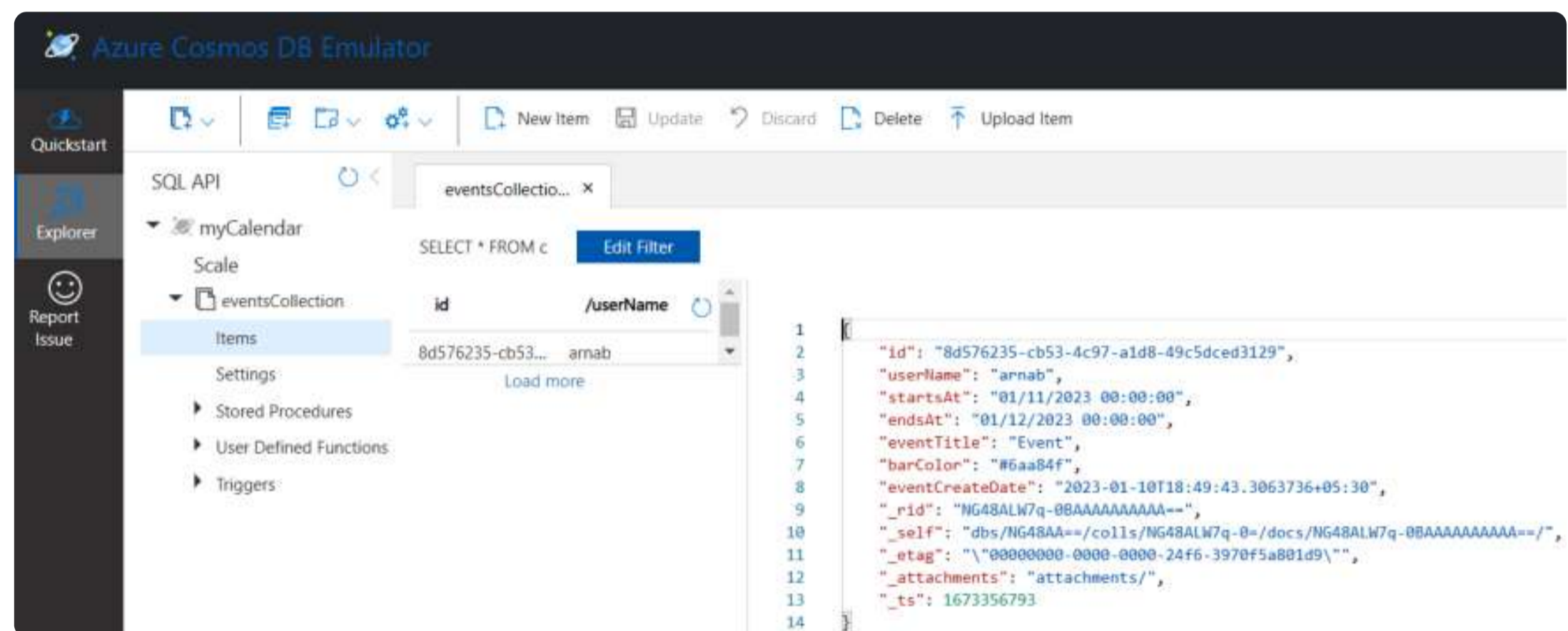
instance, one can do complex stuff with Cosmos DB in Static Web Applications / Functions. Here, `CosmosClient` object is used to configure and execute requests against the Azure Cosmos DB service. `Database` is the reference to database and `Container` a reference to container and they both are validated serverside.

`QueryDefinition` helps with the query and its parameters. `FeedIterator<>` helps in tracking the current page of results and getting a new page of results while `FeedResponse<>` represents a single page of responses which is iterated over using a foreach loop. Also notice the usage of `continuationToken` and `.WithParameter` goodness.

Though this code gets all events of a user, in production scenarios you would not wish to do that, and along with username also send start date and end date in query string to ensure limited amount of data is accessed or retrieved.

To test open a new browser in incognito / inPrivate mode, login and confirm that you are able to add new events.

To check whether the events are being persisted, check the Cosmos DB emulator explorer screen to confirm data is being persisted.



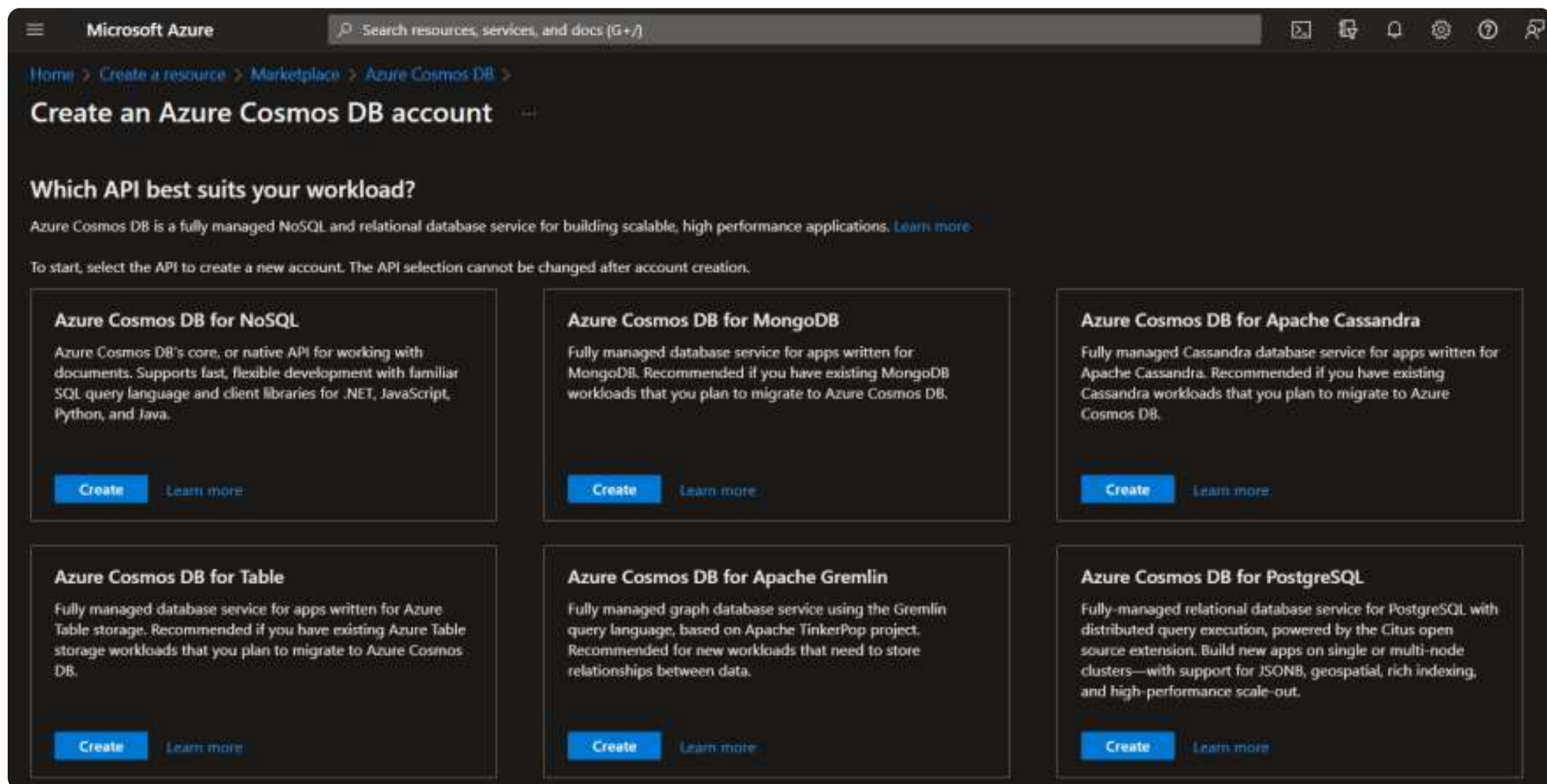
Next close the browser and open another browser again in incognito / inPrivate mode, login and confirm that you are able to view the events added earlier and persisted in Cosmos DB.

Deploy application to Azure

Provision Database

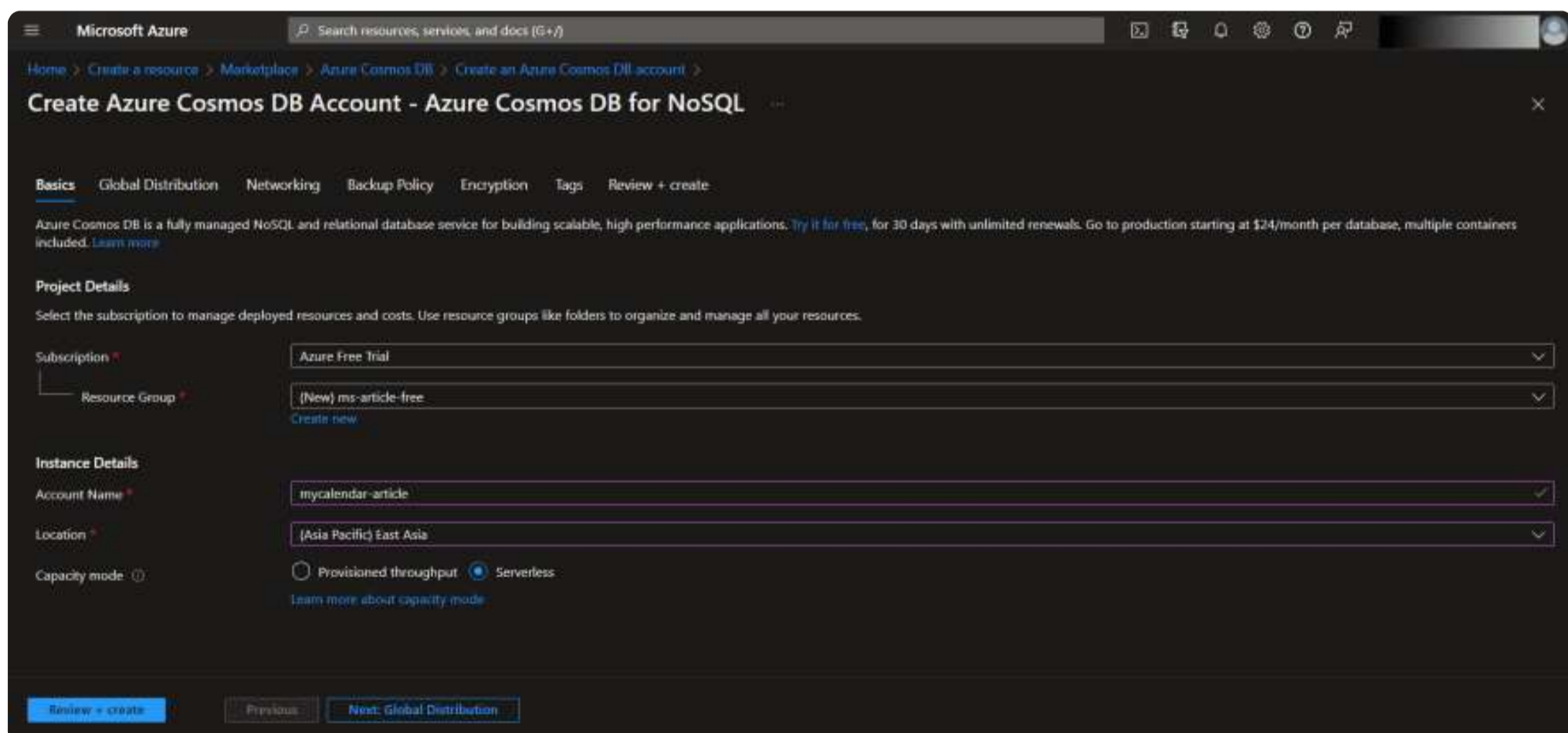
Log in to Azure Portal, Search for Cosmos DB and select the top result.

Next create Cosmos DB account by selecting *Create* under *Azure Cosmos DB For NoSQL* box.



In the next screen, select subscription, select an existing Resource Group or create a new one, add an account name, choose the nearest location and choose *Serverless* in capacity mode.

Backup policy can be changed as well and locally redundant backups selected (sufficient for POC). Selecting *Review + Create* button provisions the database.



One could also try Cosmos DB by going to cosmos.azure.com/try. Selecting the account type - *Azure Cosmos DB For NoSQL* would create a trial account for 30 days which would open in Azure Portal.

Congratulations

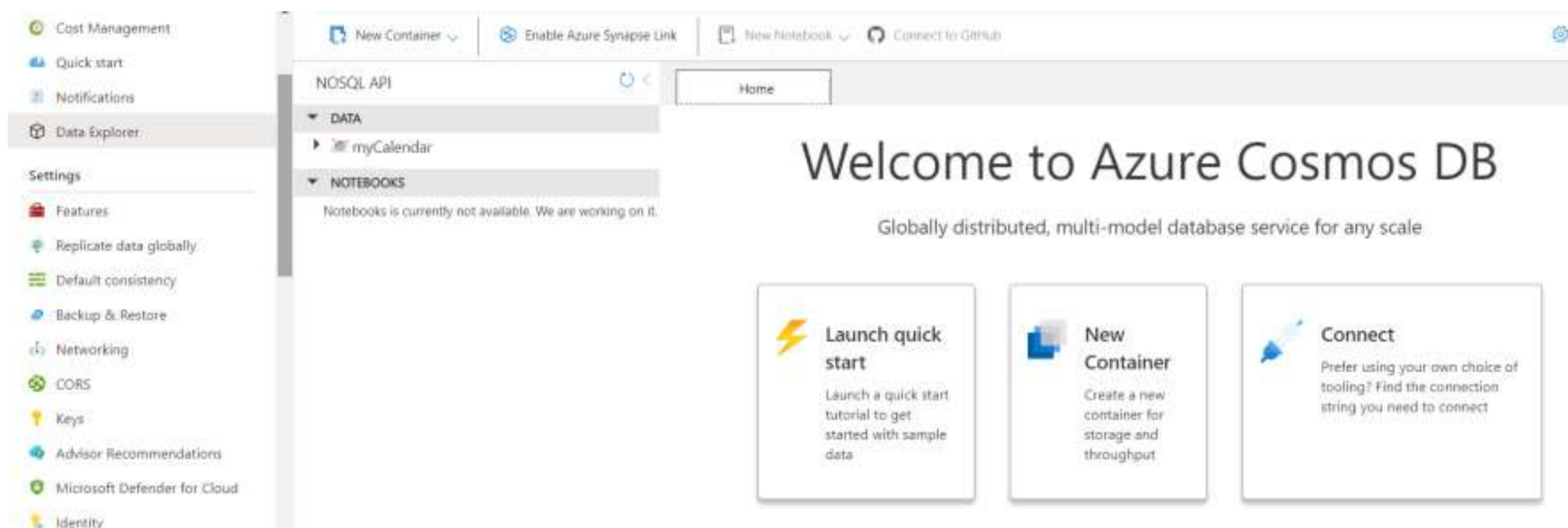
Your Azure Cosmos DB for NoSQL sandbox account is now ready for use in Azure portal.

Open it in Azure portal and follow our quick start guide to get started.

Open in portal

Once Azure Cosmos DB account page opens up, select the Data Explorer tab and a similar interface as the Data Explorer emulator opens up. Select *New Container* box, Add a new Database with name *myCalendar*, state the Container Name as *eventsCollection* and Partition Key */userName*.

Once the database is created, select the *Connect* box to view the connection string (OR select **Key** under **Settings** in left bar).

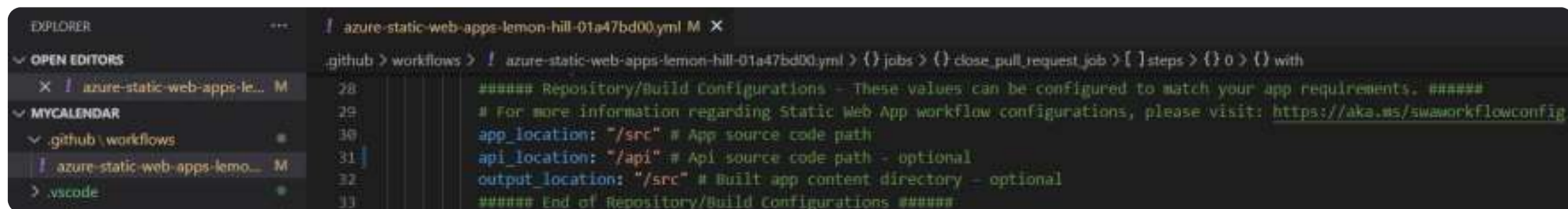


Copy the same and update the *local.settings.json* connection string in Visual Studio Code. Run the application and open it in a browser, add new events and confirm that they persisted in Azure Cosmos DB by viewing them in Azure Cosmos DB data explorer.

The CI / CD Magic

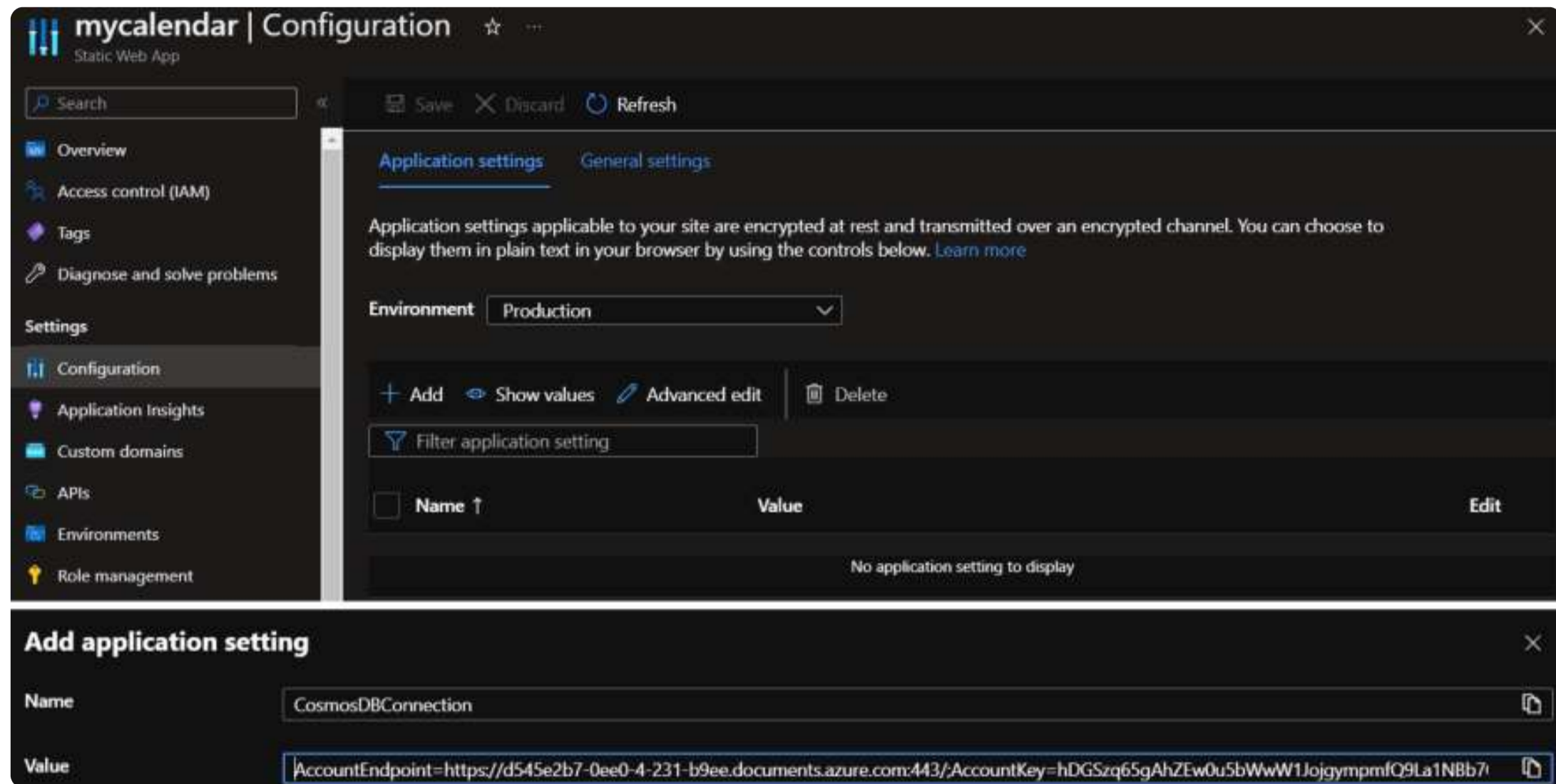
The workflow file in *.github/workflows* folder needs to be updated to let know the location of api code.

Search for **api_location** in *Repository/Build Configurations* section and update the value to **/api**.



Cosmos DB connection string was configured in *local.settings.json* file for local development. But, this file is not available in production environment.

To configure the connection string in production, go to *mycalendar* Static Web App page in Azure Portal, and select *Configuration* from the left hand bar. Click *Add* and in the next screen add *CosmosDBConnection* as Name and Azure Cosmos DB connection string as Value. Click *OK* and then *Save*.



In Visual Studio Code Terminal, open a command prompt, and from *mycalendar* folder run the following command to commit all code changes and push them to Github.

Though in this article / post, this step is done at the very end, in application development lifecycle, this ought to be done after each small change (after interface design, after authentication implementation, after API addition, etc)

```
git add -A
git status
git commit -m "Final Code with GET, POST, Auth and Azure Cosmos DB"
git push -u origin main
```

```

D:\cosmosdb\mycalendar>git commit -m "Final code with GET, POST, Auth and Azure Cosmos DB"
[main c15b8c1] Final code with GET, POST, Auth and Azure Cosmos DB
15 files changed, 686 insertions(+), 25 deletions(-)
create mode 100644 .vscode/extensions.json
create mode 100644 .vscode/launch.json
create mode 100644 .vscode/settings.json
create mode 100644 .vscode/tasks.json
create mode 100644 api/.gitignore
create mode 100644 api/GetEvents.cs
create mode 100644 api/PostEvents.cs
create mode 100644 api/Properties/launchSettings.json
create mode 100644 api/api.csproj
create mode 100644 api/event.cs
create mode 100644 api/host.json
create mode 100644 src/daypilot-all.min.js
delete mode 100644 src/styles.css

D:\cosmosdb\mycalendar>git push -u origin main
Enumerating objects: 28, done.
Counting objects: 100% (28/28), done.
Delta compression using up to 8 threads
Compressing objects: 100% (20/20), done.
Writing objects: 100% (22/22), 69.28 KiB | 2.66 MiB/s, done.
Total 22 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/c-arnab/mycalendar.git
 02020d2..c15b8c1  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.

```

And then the magic happens in the *Actions* tab in github repository.

The screenshot displays the GitHub Actions interface. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The 'Actions' tab is active, showing a list of workflow runs under the heading 'All workflows'. A search bar for 'Filter workflow runs' is present. The workflow runs table shows two runs: 'Final code with GET, POST, Auth and Azure Cosmos DB' (in progress) and 'ci: add Azure Static Web Apps workflow file' (completed). Below the table, the selected workflow 'Final code with GET, POST, Auth and Azure Cosmos DB #2' is expanded, showing a 'Build and Deploy Job' that started 1m 32s ago. The job steps are: 'Set up job', 'Build Azure/static-web-apps-deploy@v1', 'Run actions/checkout@v2', 'Build And Deploy', and 'Post Run actions/checkout@v2'. The 'Build And Deploy' step is currently active.

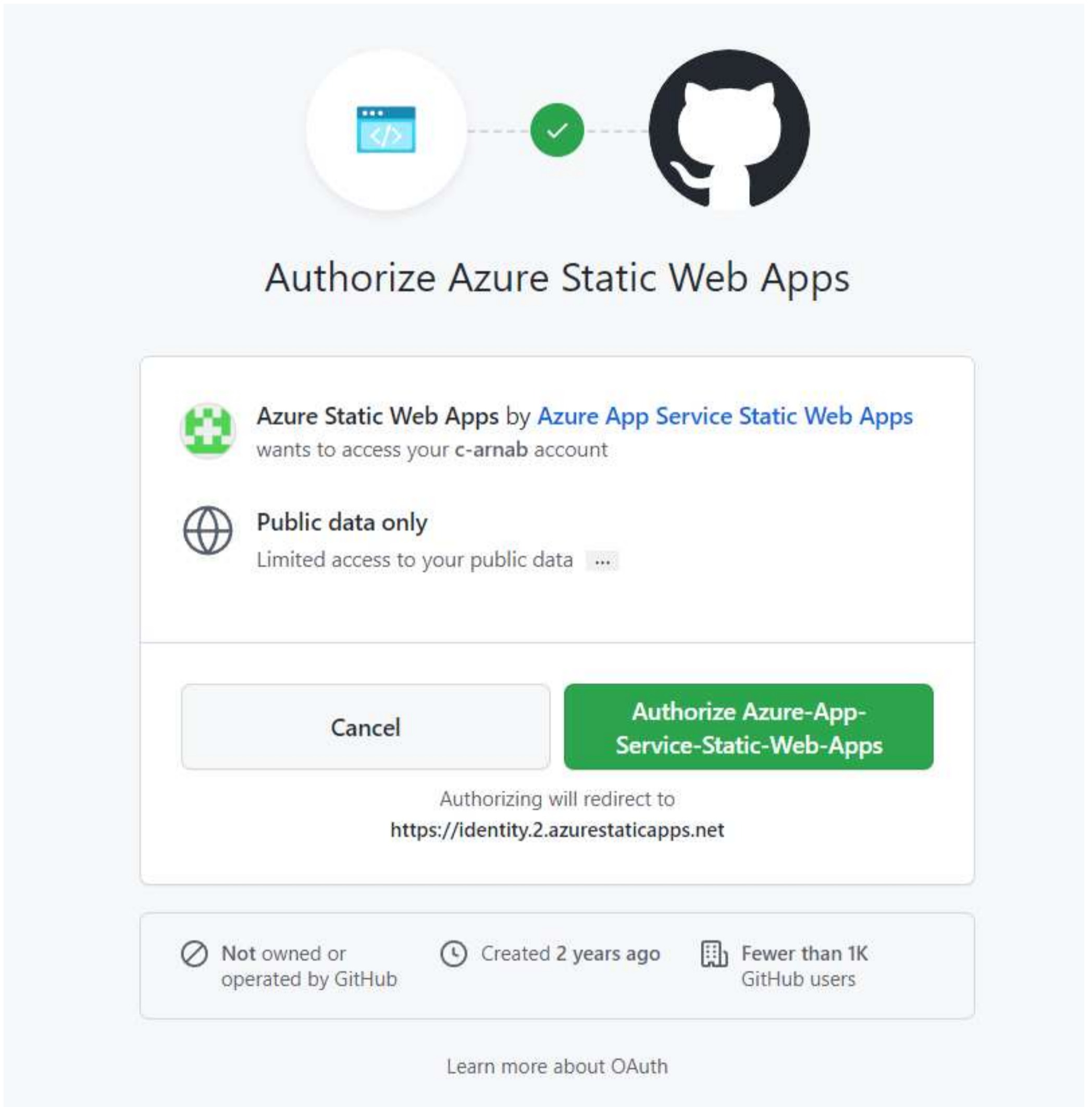
I would strongly urge all to check the steps (*Build Azure* and *Build and Deploy*) to understand all that happens automatically.

Once this step completes, the application is built on Azure and is ready for users.


Run Application


Copy the URL from *mycalendar* Static Web Apps page in the Azure Portal and paste it in a new browser window / tab.

On clicking Login, the application does not show the mock screen any more and goes to github to authenticate.






Authorize Azure Static Web Apps

 Azure Static Web Apps by [Azure App Service Static Web Apps](#) wants to access your **c-arnab** account

 **Public data only**
Limited access to your public data [...](#)

Authorizing will redirect to
<https://identity.2.azurestaticapps.net>

 Not owned or operated by GitHub  Created 2 years ago  Fewer than 1K GitHub users

[Learn more about OAuth](#)

Then there is a consent screen from Microsoft and then the Login steps are complete. Add new events, confirm that they persisted (Azure Cosmos DB data explorer) and then open the page in another tab to check the loading of existing events.

My Calendar

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
January 1	2	3	4	5 AAD Interview with Grish C	6	7
8	9	10	11	12	13	14
15	16 React Interview with Shirish Kumar	17	18	19	20	21
22	23	24	25	26	27	28
29 Kubernetes Interview with Ramesh Bhat	30	31	February 1	2	3	4

My Calendar

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
January 1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29 Kubernetes Interview with Ramesh Bhat	30	31	February 1	2	3	4

Text
AAD Interview with Grish C

Start
1/5/2023 11:00 AM

End
1/5/2023 12:00 PM

Color
Yellow

OK Cancel

Challenges

Deploying a modern web application ain't easy.

Even though in this article / post, vanilla javascript is used (to ensure that this article is useful to all developers with any framework skills such as angular, react, svelte, etc.), in production use cases some framework would be used. This application will have to be built and bundles generated.

While building the application all routes must be carefully configured to ensure users do not receive 404 errors.

APIs would have to be built and it is very much possible that different APIs are built using different technologies / languages.

User authentication will have to be implemented as well as APIs secured.

So, 'multiple servers' or even 'a cluster of servers' would have to be setup (to ensure reliability and availability) to host these bundles and APIs and if the bundles and APIs are hosted on different servers, CORS would have to be implemented or a Reverse Proxy configured. Also, a global CDN for the frontend bundles will be needed.

SSL will have to be configured as well as the ability to have custom domains added.

When maintainability of the application over a period of time is taken into account, a staging environment (similar to production) as well as an automated build process is necessary as well.

Earlier even if one used cloud services, s/he would have to do all these.

Azure Static Web Apps takes care of all the above challenges and ensures that the development team can focus on business requirements.

Business Benefit

The application developed can be used as a base for any scheduling application. Also, any application built on the lines mentioned in the article / post using Serverless technologies like Static Web Apps and Cosmos DB is beneficial as no necessity of server management, charges for consumed storage only, better scalability, lower latency, quick updates, IDE support (VS Code and SWA CLI), etc all lead to accelerated innovation.

Final Notes

Serverless on Azure - <https://azure.microsoft.com/en-in/solutions/serverless/>

Azure Static Web Apps - <https://azure.microsoft.com/en-gb/products/app-service/static>

Azure Cosmos DB - <https://azure.microsoft.com/en-in/products/cosmos-db/>

SWA Templates - <https://github.com/staticwebdev>

DayPilot Lite library - <https://javascript.daypilot.org/download/>

Source Code Repository - <https://github.com/c-arnab/mycalendar>

Top comments (0) 

[Code of Conduct](#) • [Report abuse](#)

Did you enjoy this post?