USING GOOGLE CLOUD TO HOST .NET APPLICATIONS

GitHub Source Url:
https://github.com/Magenic/MyVoteGC

March, 2017
In modern applications it is a common practice to host code in a cloud environment as well as using services and software that exist in a cloud infrastructure. The ability to remove the need to procure hardware and software as well as maintain these assets provides reduced cost. Given the strong relationship with .NET and Microsoft, developers that work on applications based on the .NET framework and related components typically run their applications in Azure. However, Google also provides a number of features that a .NET developer can use to run their applications within Google Cloud. In this while paper, we will cover how we took a .NET application that is currently hosted in Azure and moved it over into Google Cloud. We will describe what the current application does and its architecture. We will also discuss each layer of the application and what we did in Google Cloud to provide the exact (or similar) capabilities. At the end, the reader will see that Google Cloud is a viable, vibrant cloud platform for .NET applications that should strongly be considered to host their applications.

Who is Magenic?

Magenic is a leader in business technology and digital transformation consulting. Magenic understands the barriers to innovation companies are facing and applies the right technology to transform their business; including cloud computing development with leading Cloud providers such as Microsoft Azure, Google Cloud Platform and Amazon Web Services. Historically, Azure features a familiar stack of Microsoft technologies such as the .NET framework and other cloud-specific products, making Magenic's extensive history of custom application development on the Microsoft platform invaluable. Magenic has architected, designed and implemented cloud-native solutions using both Azure and other cloud platforms to deliver fast, scalable applications for their clients.

What is MyVote?

Let's take a moment to talk about the application that will be moved to Google Cloud. It's called MyVote – it's an application that lets users create polls that other users can vote and comment on. This application was created by Magenic for their Modern Apps Live conference at Live360, a developer conference hosted by 1105 Media. The intent of the application is to be a vehicle to demonstrate features, tools and platforms that developers should consider when they are building and deploying applications. We’ll cover specific architectural aspects in the next section; for now, we’ll do a quick tour of MyVote to see how it works.
Figure 1 shows the first screen in MyVote. It allows the user to log in via an account they have on Twitter, Facebook, Microsoft or Google:
After the user log in, they are routed to the main screen that displays different polls and what categories they are in as demonstrated in Figure 2:

Figure 2. Current polls in MyVote
Figure 3 illustrates how a user can select a poll to vote on it:

Figure 3. Voting on a poll
A user can also press the “Add a Poll” button to create a new poll, shown in Figure 4:

![Create A New Poll](image)

**Figure 4. Creating a new poll**

Now that we have a high-level understanding of what the application does, let's drill into specific architectural aspects of MyVote.
Current Architecture

At its core, MyVote is a .NET-based application delivered in multiple front-end applications. This is shown in Figure 5:

![MyVote Architecture Diagram](image)

**Figure 5. MyVote architecture**

Note that there are a number of UIs available that a user can choose from: a web client written in Angular, mobile applications for iOS, Android, and Windows Phone, and a UWP/Windows Store application for Windows 10 users. However, in this paper, we’ll be focusing specifically on the web client application.

The specific physical layers and services used are as follows:
- Database – MyVote uses Azure SQL Database to store poll information
- Images – Any images associated with a poll are persisted to Azure Blob Storage
- Application Server – The business logic, exposed via a CSLA DataPortal and a REST API, is hosted in an Azure Web App
- Web Server – The web application is hosted in an Azure Web App

MyVote also uses Azure App Service (formerly known as Azure Mobile Service) to authenticate users. As long as a user has an account from Microsoft, Google, Facebook or Twitter, MyVote will leverage that login via OAuth and let them create, vote and comment on polls. The reporting/BI facets use Power BI Desktop and the Power BI Service (note that the reporting aspects are not a concern of this paper).
It’s clear that MyVote is a modern application hosted in the cloud that takes advantage of Azure in many ways. Moving it to Google Cloud means that careful consideration was needed to ensure that current features would still work the same way as they do in Azure as well as take full advantage of what the Google Cloud Platform has to offer. In the next section, we’ll cover this migration in detail.

Moving to Google Cloud

So, what does it take to host a .NET application in Google Cloud like MyVote? Before we address specific implementation details, let’s cover the overall migration plan. This will provide a clear picture as we cover each layer.

Considerations and Action Plan

The current implementation of MyVote uses a number of Azure features and hosting options that must be adopted such that it can be hosted in Google Cloud. Here is a list of these changes:

- The database would be hosted in a VM hosted by Google Cloud Platform
- The application server would be converted to .NET Core and then hosted in Docker so it can be run in App Engine Flex (this isn’t required -- GCP hosts Windows Server VMs just fine as you’ll see -- we simply took this as an opportunity to showcase some of ASP.NET Core new capabilities, mainly cross-platform runtime)
- Image persistence would use Google Cloud Storage
- OAuth User Authentication would use Firebase Authentication
- The website would be hosted using a VM hosted by Google Compute Engine

The following sections will cover each of these points in detail.

Database Layer

The first step was to get our database schema hosted in the Google Compute Engine (Google’s Virtual Machine host). We decided to use a VM with SQL Server 2014 installed. Note that at the time of this project work, the ability to use a pre-defined image using SQL Server 2016 was not available, but that is now an option that developers can use (see https://cloud.google.com/compute/docs/images for details).

Creating a VM is a straightforward operation. A developer navigates to the “Compute Engine” section under the Google Cloud Platform console, and then selects “Create Instance”. Figure 6 shows the options available when creating a VM.
At this point, the developer used the “Change” button to select a specific application image, which is illustrated in Figure 7.

Once the image was created, the developer used Remote Desktop to access the VM, and proceeded to set up SQL Server with the proper credentials and security parameters.
The last step was to deploy the database to this new image. This was a relatively easy task to do, as MyVote has a database project in the Visual Studio solution. Figure 8 shows where the Publish command exists for a database project.

Figure 8. Publishing a database from Visual Studio

Overall, getting the database VM setup and the database deployed was a fairly easy process. Using a VM isn’t ideal as now we own the upkeep on that VM (and in the future we’d look to go with other hosting approaches) but the process itself was straightforward.

Middleware Layer
There were essentially two tasks to finish for MyVote’s middleware. First, we decided that the code should target .NET Core. This was done for a couple of reasons:

• There have been significant performance and architectural improvements in ASP.NET Core and we wanted to take advantage of these changes
• By moving to .NET Core it enabled us to go the Docker route (see next paragraph)

Second, the resultant executable code would be deployed in Docker. Technically, this wasn’t a requirement as we could have created a number of VMs to host our middleware and managed with **Instance Groups**. However, going the container route makes it easier to scale and maintain over time. Furthermore, this gives us the flexibility to deploy our middleware easier on other cloud platforms like Azure and AWS if we want.
The first task was primarily a migration one as .NET Core is a substantial change from traditional .NET development that targets the full .NET Framework. Fortunately, all of the NuGet packages that MyVote uses had .NET Core versions:

- Entity Framework
- CSLA.NET
- Autofac
- xUnit
- Fluent Assertions

However, it’s not possible to use the current projects and “upgrade” them to target .NET and ASP.NET Core. The recommended approach is to create new projects and move code from the old project into the new one. Therefore, our Entities, BusinessObjects and AppServer projects have “Core” projects that use code from the original .NET Framework-related project as a starting point, and then modified as needed to work in .NET Core.

Furthermore, our Entities project, which contains all of our Entity Framework-based code, was generated using a EDMX model. This no longer exists in EFCore, so an alternative approach to generating the entities was needed. Fortunately, EFCore provides the Scaffold-DbContext tool, which can generate entities and a DbContext class for a given schema. This made it straightforward to reuse our queries that were used in MyVote.

Once the code was converted and all tests passed, the Docker container could be created. For MyVote, a base image from Microsoft which contains the .NET Core command line utilities was used from Docker Hub. This image was configured to run MyVote’s web application as well as expose required ports for the outside world to communicate to the image. Detailed information on setting up and configuring a container can be found on docker.com. Here’s the Dockerfile code to create MyVote’s container:

```
FROM microsoft/dotnet:1.0.0-preview2-sdk
MAINTAINER jdpohl789@gmail.com
EXPOSE 8080
ENV ASPNETCORE_URLS http://0.0.0.0:8080/
RUN mkdir /src
COPY src /src
RUN cd /src && dotnet restore
```
The last step was to deploy our container. To do this, we needed to configure App Engine Flex so we could host our container. This was done by running a couple of commands in GCP’s console window. Figure 9 shows where the icon is that will launch the console window – it should be at the upper-right portion of the GCP Home page:

![Figure 9. Launch the Cloud Shell window.](image)

When that is clicked, a shell is launched that a developer can interact with.

To install the container, we needed to create a basic app.yaml file that would be used by the gcloud commands that would deploy our container to App Engine Flex:

```yaml
runtime: custom
vm: true
automatic_scaling:
  min_num_instances: 1
health_check:
  enable_health_check: False
```

Then we ran the following commands in the cloud shell (note that `project-id` would be replaced with the actual project ID for MyVote):

```bash
sudo docker build -t gcr.io/project-id/myvote .
gcloud docker push gcr.io/project-id/myvote
gcloud app deploy --image-url gcr.io/project-id/myvote -q
```
At this point, our container was up and running, and our REST API endpoints were live. Figure 10 shows our new instance running in App Engine Flex from the main Dashboard page:

![Dashboard showing a running instance](image)

**Image Persistence**

MyVote needed an option to store binary data for images that were used in a poll. Google Cloud has a storage option that provided this functionality and it’s called Google Storage. To use Google Storage, one must create these logical containers called “buckets”. To create a new bucket for storage, a developer navigates to the “Storage” and selects “Create Bucket”. Figure 11 shows the options available when creating a bucket.

![Creating a bucket for blob storage](image)
Once the bucket is created, a developer can use the Google.Storage NuGet package (https://github.com/GoogleCloudPlatform/google-cloud-dotnet#google-cloud-storage) to retrieve and save blob information. In the Google Storage API (beta2), the StorageClient class has a Create() method that can be used to allow a developer to specify where a given stream should be persisted. This is essentially the path MyVote needed to persist an image. Here’s the POST method for storing images for a poll in essentially 3 lines of code!

```csharp
[Authorize]
[HttpPost]
public async Task<IActionResult> Post(IFormFile imageFile)
{
    var uploadStream = new MemoryStream(new BinaryReader(imageFile.OpenReadStream()).ReadBytes((int)imageFile.Length));
    await StorageClient.Create().UploadObjectAsync(iconfig.Value.Name, imageFile.FileName, imageFile.ContentType, uploadStream);
    return new OkObjectResult(new { imageUrl = "https://storage.googleapis.com/" + iconfig.Value.Name + "" + imageFile.FileName });
}
```

One aspect to keep in mind when developing applications using services hosted in the cloud is to define abstractions around specific implementations. MyVote was originally designed to use Azure-based services and functionality, so the developers used APIs for Azure Storage directly to talk to these services. Moving to GC meant we needed to change our controllers to directly use Google Cloud Storage. A better design would have been to provide a generalized interface for image persistence and inject the implementation to the controller. As more applications are deployed into the cloud and developers investigate multiple cloud implementations, it is advised that developers shield themselves from tightly coupling their code to specific cloud-based services. Using the approach of abstraction and injection provides an application with flexibility and portability to move to different cloud providers.

**Authorization**

Google has support for social network authentication via Firebase Authentication, which would’ve been a quick way to get what we needed here except that it doesn’t support Microsoft Accounts. For the purposes of this paper, we’ve decided to leverage Firebase Authentication and support Google, Facebook and Twitter accounts and leave off Microsoft accounts. For those who would want support for all four, a customized solution would need to be developed, which then leaves the developers in a mixed world of a provided solution maintained by Google, and a customized one from the developer team. We felt this was going to be a less than ideal situation, and one would hope that Firebase Authentication would at some future point includes support for all four social networks to address this discrepancy.
To use Firebase Authentication, we simply incorporated the sample code and instructions given at the Firebase Authentication documentation site. With the exception of Microsoft Accounts support, we found Firebase Authentication to be on par with what Azure App Service provided us in terms of OAuth User Authentication functionalities.

Token authentication has quickly become a de-facto standard for modern single-page applications (SPA) and mobile applications. For this reason, MyVote leverages this scheme for securing the application from unauthorized access. In the previous version of ASP.NET, the ability to protect routes with Bearer header JSON Web Tokens (JWT) are built-in. However, in the 1.0 release version of ASP.NET Core, token validation is carried forth but the ability to generate tokens is not, leaving us with either incorporating one of few available Open Source library or develop a simple token generation mechanism on our own. Fortunately, the structure of JWT as well as the signing algorithm are well documented thus developing something that is compliant with these standards is quite manageable. The hope is Microsoft will reintroduce this functionality in a future version of ASP.NET Core and we can simply remove our code when the functionality is made available.

Our custom token generation is handled in GenerateToken() from AuthController:

```csharp
private string GenerateToken(string username)
{
    var now = DateTime.UtcNow;

    // Specifically add the jti (nonce), iat (issued timestamp), and sub (subject/user) claims.
    // You can add other claims here, if you want:
    var claims = new Claim[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, username),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        new Claim(JwtRegisteredClaimNames.Iat, ToUnixEpochDate(now).ToString(), ClaimValueTypes.Integer64)
    }
    ;

    var signingKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(Constants.SecretKey));
    var expiration = TimeSpan.FromMinutes(60);
    // Create the JWT and write it to a string
    var jwt = new JwtSecurityToken(issuer: Constants.TokenIssuer,
                                   audience: Constants.TokenAudience,
                                   claims: claims,
                                   notBefore: now,
                                   expires: now.Add(expiration),
                                   signingCredentials: new SigningCredentials(signingKey, SecurityAlgorithms.HmacSha256));

    var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);
}"
```
var response = new
{
    access_token = encodedJwt,
    userName = username,
    expires_in = (int)expiration.TotalSeconds
};

// Serialize and return the response
return JsonConvert.SerializeObject(response, new JsonSerializerSettings {
    Formatting = Formatting.Indented });

Web Application Hosting
The last piece is hosting the web application and its related assets (e.g. HTML, JS and CSS files). The simplest choice was to create another VM and host the application under IIS. For the VM creation, we recommend first searching in the Cloud Launcher portal to see if there is an existing VM image Google has stamped and approved. And in our case, we needed an Windows Server image with ASP.NET Framework support, and that is precisely one of the workload type provided by Google:

Figure 12. Creating a VM via Cloud Launcher

Once the VM image is located, we simply follow the instructions given by the portal, and within minutes the new image was created and up and running.
For deploying our code onto the newly created image, our task was greatly simplified by using the new *Cloud Tools for Visual Studio*. By downloading and installing the tool, we were able to quickly browse and download a Publishing Settings file for the new VM:

![Google Cloud Explorer](image.png)

*Figure 13. Using the Google Cloud Explorer to download a publishing settings file*
Once the publishing settings file is saved and imported to the project, we were able to deploy the code onto the new VM with great deal of ease and convenience.

We also decided to host our web application with a DNS. That way we didn’t have to refer to specific IP addresses in our test code or concern ourselves if these addresses changed for whatever reasons. This is rather straightforward to do in GCP. Under Settings, there’s a Custom domain section, which lets you add a custom domain as shown in Figure 14:

![Google Cloud Platform Custom domain Settings](image)

**Figure 14. Adding a custom domain**
Note the Webmaster Central link. This page must be visited such that the DNS is verified. This requires some setup with the provider used to register the DNS, and this setup process will vary with each provider. However, once the side identification is added via a TXT record, Google can verify the site, and then the page can be referenced with the DNS. Figure 15 shows this verification page:

**Figure 15. Verifying the DNS entry**

Performance Considerations
One question remains: are there differences between Azure and Google Cloud in terms of performance? Magenic's QA team ran a number of performance tests comparing MyVote in Azure and GCP. These tests are based on the Selenium framework to execute tests against the web views along with tests that exercise the REST APIs directly. The tests were executed against the Azure and GCP instances. These tests were executed multiple times and their averages were calculated. The results showed that the application worked in GCP just as well as it does in Azure, if not slightly faster. Here's a list of the tests, their averages and the differences between Azure and GCP:
<table>
<thead>
<tr>
<th>Test Name</th>
<th>Azure Averages</th>
<th>Google Averages</th>
<th>Percentage Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests.PollTests.VerifyHomepageIconFromPoll</td>
<td>0.31</td>
<td>0.36</td>
<td>-15%</td>
</tr>
<tr>
<td>Tests.PollTests.VerifyLogoutFromPoll</td>
<td>0.22</td>
<td>0.30</td>
<td>-26%</td>
</tr>
<tr>
<td>Tests.PollTests.AddPoll</td>
<td>4.11</td>
<td>3.56</td>
<td>15%</td>
</tr>
<tr>
<td>Tests.LoginTest.NewLogin</td>
<td>6.07</td>
<td>6.09</td>
<td>0%</td>
</tr>
<tr>
<td>Tests.LoginTest.TwitterLogin</td>
<td>3.58</td>
<td>3.30</td>
<td>8%</td>
</tr>
<tr>
<td>Tests/LoginTest/FacebookLogin</td>
<td>3.59</td>
<td>3.20</td>
<td>12%</td>
</tr>
<tr>
<td>Tests/LoginTest/GoogleLogin</td>
<td>4.64</td>
<td>3.88</td>
<td>20%</td>
</tr>
<tr>
<td>Tests/LoginTest/BadCredentialTest</td>
<td>3.71</td>
<td>3.51</td>
<td>6%</td>
</tr>
<tr>
<td>Tests.Services/Respond/GetInvalidSinglePoll</td>
<td>0.07</td>
<td>0.04</td>
<td>57%</td>
</tr>
<tr>
<td>Tests.Services/Respond/PutSinglePollServerError</td>
<td>0.15</td>
<td>0.09</td>
<td>62%</td>
</tr>
<tr>
<td>Tests.Services/Polls/GetListOfPolls</td>
<td>0.07</td>
<td>0.05</td>
<td>36%</td>
</tr>
<tr>
<td>Tests.Services/Polls/GetSinglePoll</td>
<td>0.11</td>
<td>0.08</td>
<td>29%</td>
</tr>
<tr>
<td>Tests.Services/Polls/GetPollsByMostPopular</td>
<td>0.07</td>
<td>0.05</td>
<td>40%</td>
</tr>
<tr>
<td>Tests.Services/Polls/GetPollsByNewest</td>
<td>0.06</td>
<td>0.06</td>
<td>4%</td>
</tr>
<tr>
<td>Tests.Services/Polls/GetSinglePollServerError</td>
<td>0.06</td>
<td>0.05</td>
<td>26%</td>
</tr>
<tr>
<td>Tests.Services/Polls/GetSinglePollNotFound</td>
<td>0.07</td>
<td>0.04</td>
<td>69%</td>
</tr>
<tr>
<td>Tests.Services/Polls/PutSinglePollBadRequest</td>
<td>0.14</td>
<td>0.08</td>
<td>80%</td>
</tr>
<tr>
<td>Tests.Services/Polls/DeleteSinglePollServerError</td>
<td>0.07</td>
<td>0.05</td>
<td>36%</td>
</tr>
<tr>
<td>Tests.PortalPollsPageTests/SortByMostPopular</td>
<td>2.45</td>
<td>2.45</td>
<td>0%</td>
</tr>
<tr>
<td>Tests.PortalPollsPageTests/SortByNewest</td>
<td>2.43</td>
<td>2.41</td>
<td>1%</td>
</tr>
<tr>
<td>Tests.PortalPollsPageTests/LogoutFromPolls</td>
<td>0.23</td>
<td>0.21</td>
<td>8%</td>
</tr>
<tr>
<td>Tests.PortalPollsPageTests/ConfirmHomepageIconReturns</td>
<td>0.35</td>
<td>0.33</td>
<td>4%</td>
</tr>
<tr>
<td>Tests.PortalPollsPageTests/VerifyAddPollButton</td>
<td>0.34</td>
<td>0.31</td>
<td>6%</td>
</tr>
<tr>
<td>Tests.PortalPollsPageTests/OpenAnExistingPoll</td>
<td>0.21</td>
<td>0.21</td>
<td>3%</td>
</tr>
</tbody>
</table>
Figure 16 is a graph of this data:

![Bar chart comparing Azure and Google Cloud performance](chart16.png)

**Figure 16. Comparing application performance between Azure and Google Cloud**

As Figure 16 shows, with each feature in MyVote the differences in speed were generally identical across the board. The UI tests did not show any substantial differences between the two implementations (0.3%), though the REST API tests were 43% faster in GCP than in Azure. Note: This isn’t entirely an apple-to-apple comparison as the code in Azure was not changed to use our Docker container and is still running against the full .NET Framework.
Conclusion

In this paper, we took a .NET web application - MyVote - that was running in Azure and moved it into Google Cloud. We discussed the various changes needed so MyVote would not only execute correctly in the Google Cloud, but maximize its performance and maintenance by leveraging various platform offerings created by Google for the .NET development community. These changes centered on VM creation, image persistence and authorization. In the end, the amount of code that needed to change in the application related to features was minimal with similar performance characteristics.

If you’re writing .NET applications that are hosted in the cloud, GCP is an option that .NET developers should strongly consider. GCP has services and capabilities that are similar to Azure, with an intuitive UI and platform that allows developers to quickly setup and deploy VMs and applications.

About Magenic

Magenic is a leader in business technology consulting. We understand the barriers to innovation companies are facing and apply the right technology to transform their business.

Visit us at magenic.com or call us at 877.277.1044 to learn more or to engage Magenic today.