Google Cloud Platform

Modernizing Your .NET Application for Google Cloud Platform

by Andy Wu, Solutions Architect, Magenic

Table of Contents

Executive Summary

As competitors adopt new technologies to streamline processes and attract customers, CIOs are under increased pressure to ensure that their organizations remain competitive by enabling their IT to develop and iterate faster than their competition. Unfortunately, most legacy enterprise applications are ill-suited to respond to this pressure. In the worst case, they might even be a barrier for the company to move their business forward. Doing nothing is not an option as systems will continue to collect and compound technical debts, resulting in even longer and more challenging system upgrade cycles.

As the second of this series on taking the cloud-native journey, we will continue the effort by effectively applying many of the more popular application modernization techniques, such as database migration with managed services, integration with SaaS based offerings, distributed caching and framework updates to our application: Microsoft .NET PetShop. Here is the GitHub repository for the project.

Motivations for Application Modernization

Organizations modernize applications for a variety of reasons, but the recurring themes seem to be:

- Reducing cost and complexity
- Increasing speed and agility by lowering the barrier to innovation and building new features

Everyone has heard the old adage “if it ain’t broke, don’t fix it”. Unfortunately, this does not mean that you can avoid investment in mission-critical, enterprise software because it lives in a dynamic world where customer needs and technology continue to transform and evolve. As time progresses, legacy code can cost the company in a significant fashion: either in performance, ability to innovate, operational efficiency, opportunity for lower cost structures, or unnecessary code maintenance. We will go through this concept with some specific examples in this white paper and illustrate the needs, or rather, **opportunity** for app modernization.

Strategy for Application Modernization

Lifting your critical business technologies off legacy implementations and refactoring them to take advantage of modern technologies and services is much like the undertaking when you decided to start running to improve your health. You won’t simply open the front door and run a marathon. You need to come up with the right game plan, the right foundation, and the right tools. Create a plan, iterate on it, and keep moving forward.

For this application modernization, your plan should take the following into account:

1. Retain what is strategic and differentiated
2. Update language and frameworks to the latest versions or more modern replacements that have better support
3. Migrate to cloud managed service offering wherever possible
4. Leverage the software-as-a-service (SaaS) model

The underlying concept behind this strategy is to reduce the footprint both in terms of the system's overall code base as well as operation and administration resources needed to keep the system running. Just as companies are realizing that running a data center should probably not be part of their core competency, neither should developing and maintaining any code base that doesn't offer any competitive advantage. As this concept proliferates, architects and developers are reevaluating their systems to determine whether common libraries or tooling can replace custom maintained solutions to allow them to focus on the items that provide the most business impact.

The App Modernization Roadmap

After analysing the current code base of PetShop, we discovered four areas that should be the target of our modernization effort. In the rest of this white paper, we will detail each area with the **why** in terms of benefits to be derived, as well as **how** we will make the modifications.

Authentication

The current authentication for PetShop is implemented using the ASP.NET Membership API. First introduced in 2005 with ASP.NET 2.0, this framework was quite popular when it was released, as it removed the need to develop an authentication system from scratch. However, as apps and web sites strive to reach an ever expanding audience, an authentication system must keep up with its security functionality in order to properly protect both the users and the providers of the system. Features such as two-factor authentication, social network integration (Facebook, Google, etc.), smart devices support, as well as industry standard open protocols (e.g.OAuth 2.0 and OpenID Connect) are all becoming must-haves for any modern-day web applications. To develop and continually keep up with changes in this space creates a tremendous demand on resources for most corporate development organizations. As such, one should find ways to offload this responsibility.

Laurence Moroney, a Developer Advocate at Google, made a useful analogy about coding for authentication: “Building an authentication system for your app can feel a lot like paying taxes. They are both relatively hard-to-understand tasks that you have no choice but doing, and could have big consequences if you get them wrong.”

Seldom, if ever, does a company get ahead of their competition because their system has great login features. Every company has limited resources when it comes to software development; we need to prioritize these limited resources to focus on innovative and differentiating features

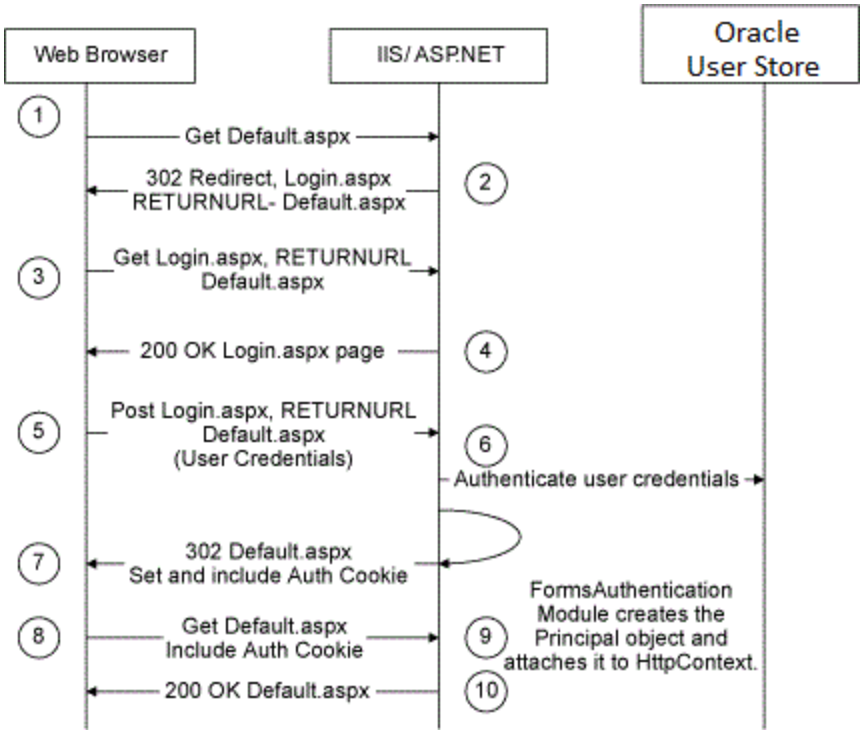
and not on reinventing common functionality. Thus this is an area we should examine to see if we can “outsource,” and leave the functionality to companies that make a business out of having a highly-secured authentication system for developers to consume.

One great choice for offloading this functionality is Firebase Authentication from Google. Among the features this service offers are: simple implementation, free, multi-platform support, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more. With the authentication service being essentially free (fee only starts when an application needs Phone Verification above 10k per month), I would think one would be hard pressed to find a better pricing plan.

Integrating Firebase with ASP.NET Forms Authentication

Now that we have identified a great managed service to outsource PetShop’s authentication, the next step is to figure out the simplest way to integrate it into PetShop. ASP.NET Membership relies heavily on Forms Authentication to process authentication. At a high level, Forms Authentication uses an authentication ticket that is automatically generated by ASP.NET once a user is successfully authenticated and logs on to a site. The forms authentication ticket is cookie based. If a user requests a page that requires secured access and that user has not been previously authenticate to the site, then the user is redirected to a preconfigured login page. This login page prompts the user to supply their credentials, which are then passed to the server and validated against a user store, such as the Oracle database used by PetShop. After the user's credentials are authenticated, the user is redirected back to the originally requested page.

Here is a pictorial view of the code flow described above:



With this understanding of the current authentication implementation, we can now embark on designing an architecture to incorporate Firebase Authentication with ASP.NET Web Forms.

Fortunately, this is greatly facilitated by Firebase providing developers with a drop-in auth solution called FirebaseUI, an open-source library that handles the Web UI flows for signing in users with any one of the following choice:

- email and passwords
- Phone numbers
- Federated identity providers, including Google, Facebook, Twitter, and Github

This library also include the code for the following authentication related use cases:

1. Sign-up and sign-in
2. Password reset
3. Prevention of account duplication
4. Account Chooser for remembering emails

Detailed documentation on how to use this open-source software (OSS) library can be found here.

Code to integrate FirebaseUI with ASP.NET

Now that we have identified FirebaseUI to be the library to handle the frontend login UI interactions, the only thing left to develop is the code that will integrate Firebase Auth with ASP.NET Web Forms. What we would like is something where we can simply “plug in” the

FirebaseUI where the current login UI takes place, and have the rest of the application continue to work as if nothing has changed. Well, I think I have just the thing for you.

As mentioned earlier, Windows Forms Authentication system relies on an auth cookie to keep track of the authenticated user of the system. This cookie is issued as part of the normal ASP.NET Web Forms Login Controls workflow, details of this can be found here. The content of the cookie is actually a ticket object of FormsAuthenticationTicket type, which can be created by specifying the version of the cookie, the directory path, the issue date of the cookie, the expiration date of the cookie, whether the cookie should be persisted, and, optionally, user-defined data. For example, something like:

```
FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(1,
        "userName",
        DateTime.Now,
        DateTime.Now.AddMinutes(30), // value of timeout property
        false, // Value of IsPersistent property
        String.Empty, //user-defined data
        FormsAuthentication.FormsCookiePath);
```

Note: Be mindful the of the user-defined data parameter, it's the key to our integration strategy.

Normally, the ticket creation process is handled behind the scene when applications are developed with the set of Login Server Controls. But for our purpose of integrating Firebase Auth and ASP.NET Web Forms, we actually need to "hijack" the ticket creation process and inject data we get from the FirebaseUI drop-in auth solution so the two systems can work together seamlessly. To do so, we first need to incorporate the assets that are part of FirebaseUI library. This can be easily accomplished by including all the Javascript and css files that are all part of the library to the appropriate page(s). In our case, the Login.aspx page below is what's needed in order to add Firebase Auth to a given ASPX page:

```
<script type="text/javascript"
src="https://www.gstatic.com/firebasejs/live/4.1/firebase.js"></script>
<script type="text/javascript" src="config.js"></script>
<script type="text/javascript" src="common.js"></script>
<script type="text/javascript"
src="https://cdn.firebase.com/libs/firebaseui/2.2.1/firebaseui.js"></script>
    <link type="text/css" rel="stylesheet"
href="https://cdn.firebase.com/libs/firebaseui/2.2.1/firebaseui.css" />
<script type="text/javascript" src="app.js"></script>
```

Note: One needs to sign up for usage of Firebase Auth by going to the Firebase Console and create a new project that will generate a new API key that one can incorporate into the config.js file.

A full tutorial on the usage of FirebaseUI SDK can be found here.

The key Javascript event to tap into for our integration is the *handleSignedInUser,* located inside the *app.js* file. Just as one expects, it is the event that gets fired when a user has successfully signed in using Firebase Auth. As part of this event, a parameter of Firebase User Account object will be passed. Among the useful properties of this object, we are interested in two in particular—the **displayName** and the **uid**, which are the Display Name of the Firebase user and a unique ID assigned by Firebase. And once we receive this info via the Javascript event, we need to "pass" them to the server end so we can manually create the FormsAuthenticationTicket object. To pass the info to the server, we simply do a redirect with query string:

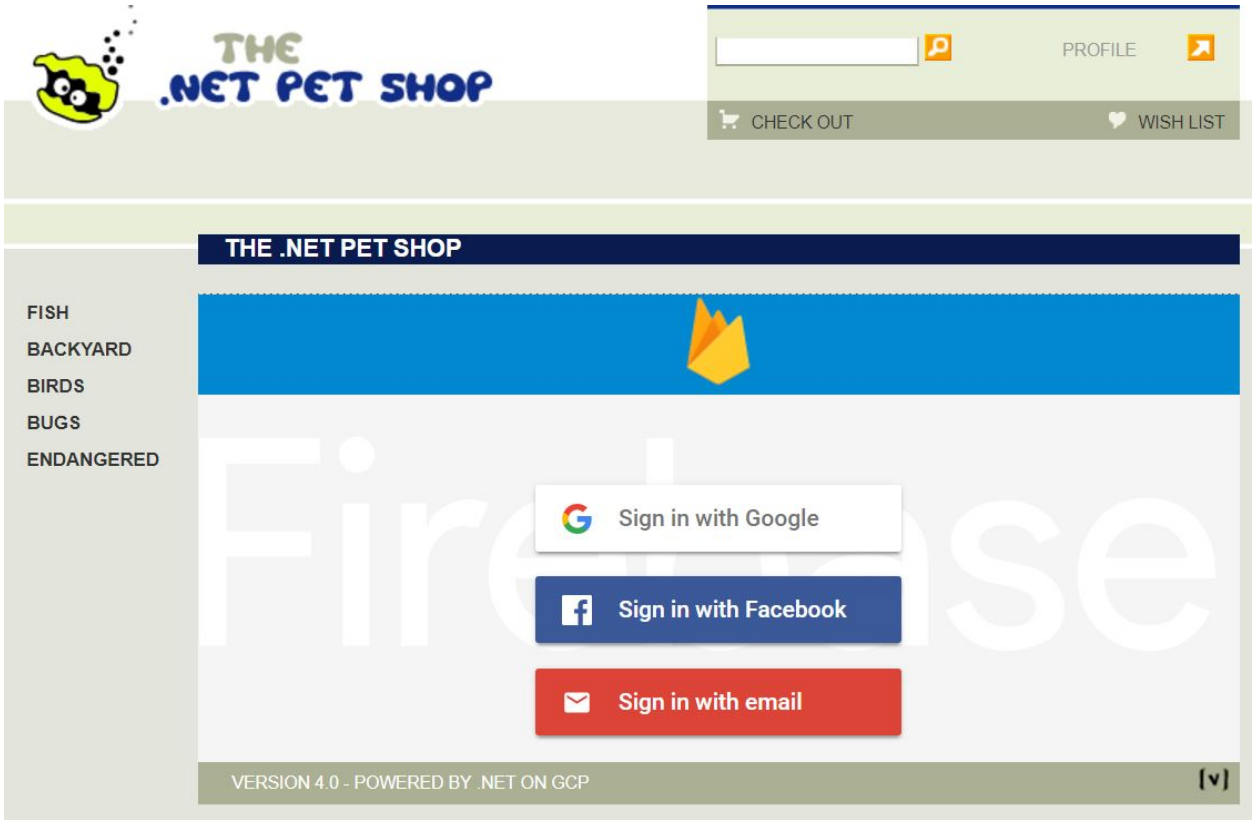
```
var handleSignedInUser = function(user) {
  var returnurl = getQueryString('ReturnUrl');
  window.location.replace("/login_fba.aspx?uid=" + user.uid + "&displayname=" + user.displayName
+ ((returnurl == null) ? "" : "&ReturnUrl=" + returnurl) );
};
```

And inside the redirected page's code behind (*login_fba.aspx.cs*), we will create the FormsAuthenticationTicket object with the info that was passed via query string:

```
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!String.IsNullOrEmpty(Request.QueryString["uid"]))
        {
            FormsAuthenticationTicket tkt;
            string cookiestr;
            HttpCookie ck;
            tkt = new FormsAuthenticationTicket(1,
                                    Request.QueryString["displayname"],DateTime.Now,
                                    DateTime.Now.AddMinutes(30), true,
                                    Request.QueryString["uid"]);
            cookiestr = FormsAuthentication.Encrypt(tkt);
            ck = new HttpCookie(FormsAuthentication.FormsCookieName, cookiestr);
            ck.Expires = tkt.Expiration;
            ck.Path = FormsAuthentication.FormsCookiePath;
            Response.Cookies.Add(ck);
            string strRedirect = Request["ReturnUrl"];
            if (string.IsNullOrEmpty(strRedirect))
                strRedirect = "default.aspx";
            Response.Redirect(strRedirect, true);
        }
     }
```

As mentioned earlier, we leveraged the user-defined parameter (highlighted above) of the FormsAuthentication constructor to place the unique ID we get from Firebase Auth. Once it's in the ticket, we can always retrieve it whenever it's needed within the PetShop workflow. That's how we got the two frameworks to integrate seamlessly!

The final product:



Database

Companies need to stretch every last bit out of their IT dollar in order to drive customer engagement and other critical initiatives. As many companies are discovering, one such area of opportunity is to transition the usage of Relational Database Management System (DBMS) from a traditional commercial vendor licensing model to an open-source software (OSS) model. While it can be the most critical and sensitive layer of the infrastructure stack, the database layer also has the potential for receiving the greatest cost savings. And it's no secret that the licensing cost for Oracle DBMS is among the highest in the industry, therefore, it would behoove companies to re-examine their practice for paying for this functionality.

PostgreSQL is a proven open-source database technology that can handle the most demanding use cases. PostgreSQL has the backing of 25 years of development refinements and was sparked by the same technological revolution as Oracle's DBMS. Both of these database engines were born out of Edgar F. Codd's groundbreaking 1970s research paper that served as the foundation for relational database management. One became the proprietary DBMS offered by Oracle while another, PostgreSQL, was developed by a small group of Massachusetts Institute of Technology (MIT) scientists dedicated to advancing the field. Now in its third decade

of development, PostgreSQL has reached a tipping point in making it a strong alternative to Oracle for many companies.

At Google Cloud Next 2017, Google announced support for PostgreSQL as part of their managed database service offering, Cloud SQL. Not only does it offer the advantage of the OSS licensing model, with Google Cloud Platform (GCP), there is no up-front commitment cost. Best of all, users still get the same billing innovation GCP is famous for, such as per-second billing and sustained use discounts, making this a great choice for companies considering making the switch.

Wait, there is more (opportunity for saving)

When companies use an on-premises database system, there are usually multitudes of administrative and maintenance tasks that come with operating such a system. Repetitive and mundane tasks such as backups, replication setup, patches, and system updates, will typically fall on shoulders of the database administrator (DBA), taking valuable time away from them to focus on more strategic activities of their role. Contrast this with a fully-managed database service like Cloud SQL, where the provider is responsible for the administrative tasks while providing you with a guaranteed service level agreement (SLA). That's a tremendous way of reducing the total cost of ownership (TCO) of operating a DBMS.

Tools and process to migrate data from Oracle to PostgreSQL

If one decides they want to transition away from an Oracle RDBMS, the next critical question should be: how does one migrate the data? After much research, we decided on another OSS tool: Ora2PG, one of the most popular tool for this task.

To start, we downloaded the tool from GitHub. Ora2PG is written mostly in Perl, therefore it requires the DBD:Oracle database connection package, available for download from the Perl user repository, CPAN. Next we built the Ora2PG tool with the downloaded source from GitHub by running the following commands:

```
perl Makefile.PL
make && make install
```

After verifying that it got built without error, we made a new project tree by:

```
ora2pg --project_base /app/migration/ --init_project test_project
```

This command creates a folder hierarchy and a project-specific config file, as well as adding a couple of helpful scripts to the project folder. The config file contains the connection details of the database server. One needs to modify the config file with the proper credentials and a user/schema that the tool will use to perform the data extraction.

To run the schema extract, we use the following shell script four times, for each of the PetShop specific Oracle Schema: MSPETSHOP4, MSPETSHOP4ORDERS, MSPETSHOP4PROFILE, MSPETSHOP4SERVICES, changing the target schema each time before we run the command.

```
./Export_Schema.sh
```

After the extraction is completed, one should have the all the schema definition needed to recreate the PetShop application tables. One could then simply execute the generated SQL file using a command line utility like psql or other GUI-based PostgreSQL query tool such as pgAdmin.

The next step is to extract the actual data from Oracle. To do that, we need to execute the following command:

```
ora2pg –t COPY –o data.sql –b $namespace/data –c
$namespace/config/ora2pg.conf
```

This command outputs a single .sql file with all of the data as COPY statements (you can set it for INSERTs as well). Here is a sample snippet of what the tool generates:

```
COPY category (categoryid,name,descn) FROM STDIN;
FISH  Fish  FISH
BYARD Backyard    Backyard
BIRDS BIRDS Birds
BUGS  Bugs  Bugs
EDANGER     Endangered  Endangered
```

With these queries in hand, we simply execute them using our tool of choice and we are done with the migration process!

A cautionary note: As it is customary with most initial data load processes, there is a high likelihood that one will need to temporarily drop the table constraints and indices before loading the data. Therefore be prepared to do so if errors are found due to table constraints and indices violations.

Caching: In-Process vs. Distributed

Caching is a commonly used and powerful technique for increasing application performance through a simple concept: instead of doing expensive work (like a complicated calculation or complex database query) every time we need a result, the system stores (caches) the result of that work and simply grabs it from the cache store the next time it is requested without the expensive reprocessing. The original developers of PetShop also understood this concept and implemented caching in PetShop using the tried and proven HttpRuntime.Cache from the .NET Framework.

HttpRuntime.Cache is an in-process caching architecture, which makes cached data access very fast. However, it is limited to a single process's memory, and is therefore not scalable beyond a single server. A distributed cache on the other hand is a scalable solution where you can always add more servers to increase cache capacity and throughput. The out-of-proc nature of distributed cache also makes it possible to access coherent cached data by other running application process(es). A good example of this use case is a web farm of front-end servers having access to the same cached data. Last, but certainly not least, a distributed cache architecture will accommodate the need for high availability in a clustered configuration. That's why, in most modern-day web applications that are expected to handle high load with high availability, distributed cache is usually a common component within the system architecture.

Redis is an open-source distributed and in-memory cache used by many companies and in countless mission-critical production environments. Redis is supported by client libraries in many programming language, and it's included in a multitude of packages for developers. Today, one would find it rare if a web stack did not include built-in support for Redis.

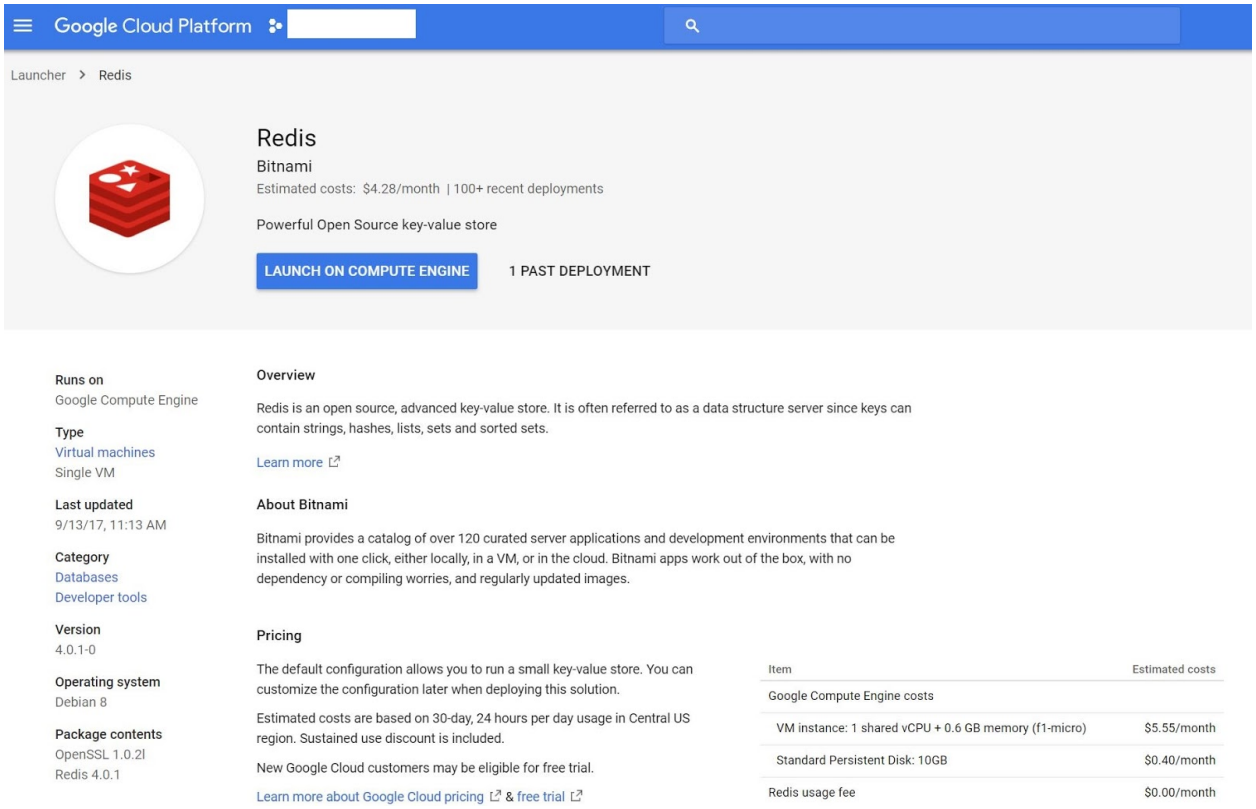
Why is Redis so popular? Not only is it extremely effective, it is also relatively simple. Getting started with Redis is considered easy work for a developer. Being one of the most popular tools in the industry, there are plenty of resources available at one's fingertips. It takes only a few minutes to set up and get them working with an application.

Deploying Redis with a few clicks

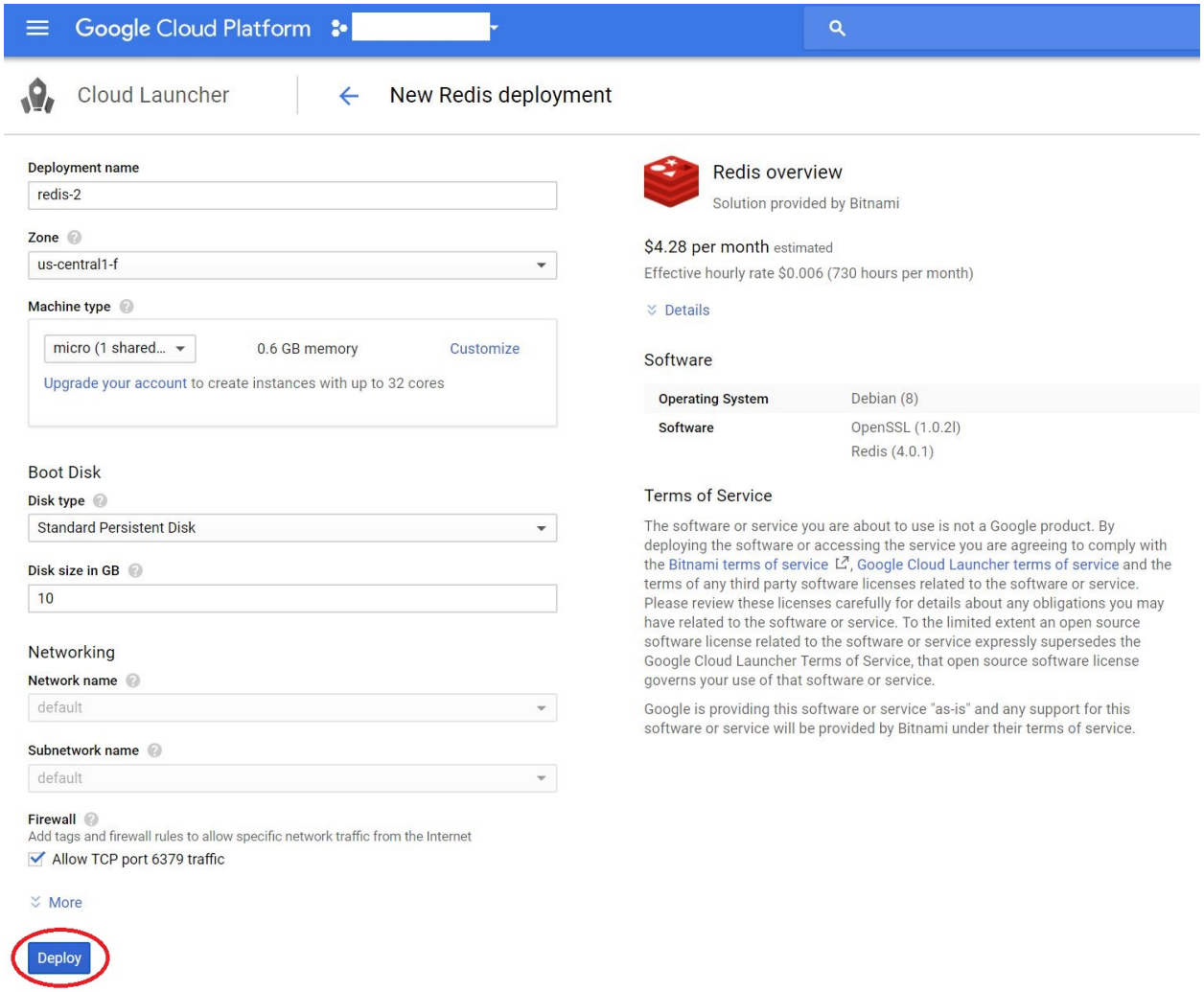
Once we determined the benefit of transitioning PetShop's caching functionality from an in-process implementation (HttpRuntime.Cache) to a distributed cache like Redis, the next step is to set up the infrastructure (i.e. the VM for hosting Redis). To do this in GCP is extremely easy. In fact, the setup of the Redis cache server VM can be accomplished with just a few clicks!

This fast setup of a Redis cache is made possible by Google's Cloud Launcher, a marketplace where you can find—and launch—more than 160 popular open source packages that have been configured by Bitnami, or Google Click to Deploy. Once the appropriate deployment package is found from the library, deploying it is incredibly straightforward: simply specify the parameters needed by the package (if any) and the package will be up and running within minutes. Cloud Launcher is designed to make developers, and operators, more efficient, removing the operational deployment and configuration tasks so they can focus on what matters: developing and running their application.

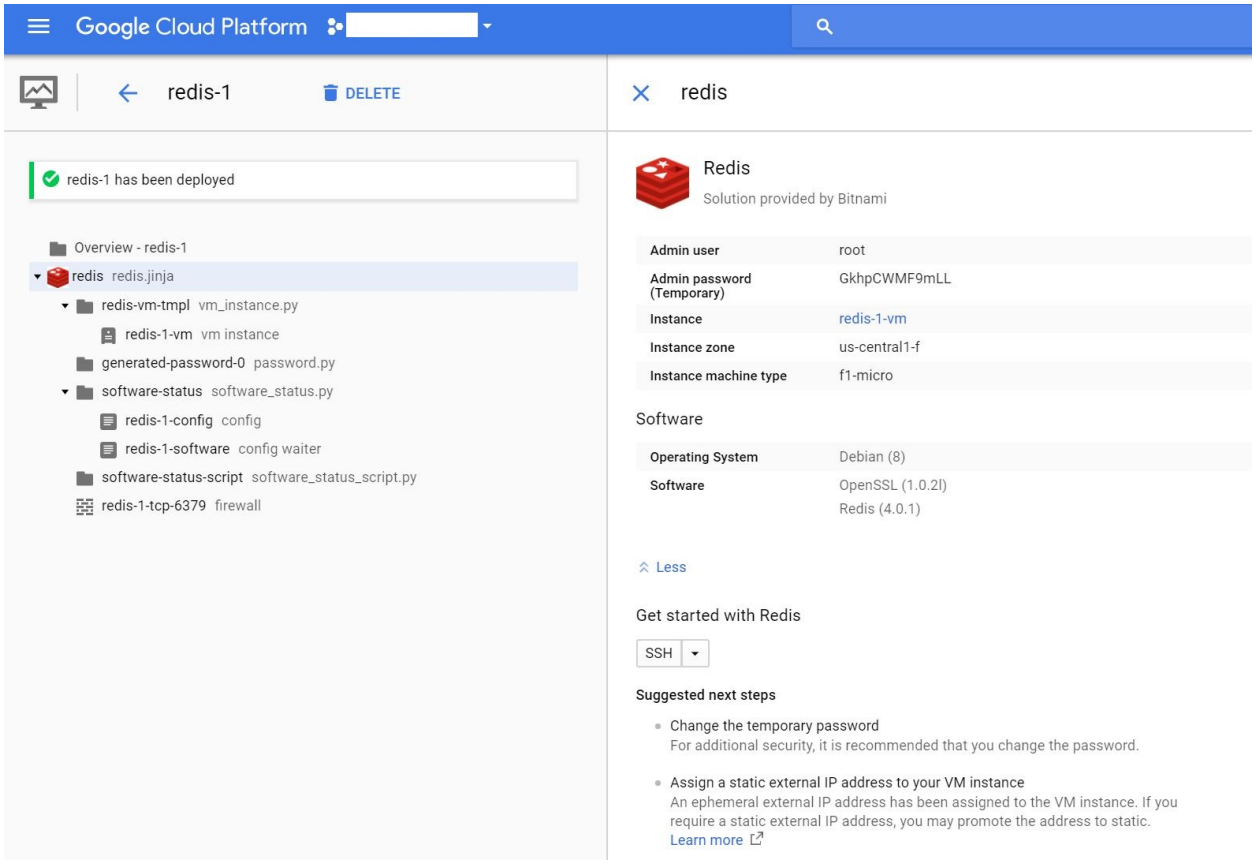
Let's walk through the example for setting up Redis with Cloud Launcher. First go to the Cloud Launcher marketplace landing page and type in the word Redis in the search box. The following search result should show up:



Click the ‘LAUNCH ON COMPUTE ENGINE’ button in the middle, and then the following parameter page should be shown:



Specify the parameters needed by the deployment, then click the ‘Deploy’ button at the bottom of the page. Within minutes, your Redis server will be ready for use:



Creating a .NET Core class to work with Redis

On the coding side of things, it is also quite simple to come up with a .NET Core class component that will work with the Redis Cache Server we have just set up. First we need to find a .NET Core-compatible Redis client package for connecting to, and communicating with, the Redis server. As a testament of the popularity of Redis, a Google search quickly revealed there are more than few available for the job. The one we finalized on is called StackExchange Redis Client, as it seems to have the most sample code that we can leverage. Once the StackExchange NuGet package is added to our project, we simply create a static class, RedisCacheManager, that contains the respective methods for:

- Initializing the connection to Redis
- Storing (caching) any object to Redis
- Getting (retrieving) any cached object from Redis
- Invalidating any cached object from Redis

```
public static class RedisCacheManager
    {
        private static ConnectionMultiplexer connection;
        private static IDatabase cache;
        private static int DefaultCacheDuration = 360;
        public static bool Initialized { get; private set; }
        public static void Initialize(string connStr, int defaultCacheDuration = 360)
```

```
    {
        try
        {
            connection = ConnectionMultiplexer.Connect(connStr);
            cache = connection.GetDatabase();
            Initialized = true;
            DefaultCacheDuration = defaultCacheDuration;
        }
        catch
        {
            Initialized = false;
        }
    }

    public static void Store(string key, object content, int? duration = null)
    {
        if (!Initialized) return;

        string toStore;
        if (content is string)
        {
            toStore = (string)content;
        }
        else
        {
            toStore = Newtonsoft.Json.JsonConvert.SerializeObject(content);
        }
        if (duration == null) duration = DefaultCacheDuration;

        cache.StringSet(key, toStore, new TimeSpan(0, 0, duration.Value));
    }

    public static T Get<T>(string key) where T : class
    {
        if (!Initialized)
            return null;
        try
        {
            var fromCache = cache.StringGet(key);
            if (!fromCache.HasValue)
                return null;

            string str = fromCache.ToString();

            if (typeof(T) == typeof(string))
                return str as T;
            return Newtonsoft.Json.JsonConvert.DeserializeObject<T>(str);
        }
        catch
        {
            return null;
        }
    }

    public static void Invalidate(string key)
    {
        if (connection != null)
            cache.KeyDelete(key);
    }
}
```

Make the case for using .NET Core

Earlier this year, Microsoft celebrated the 15 years anniversary of the release of .NET Framework. That is certainly a great accomplishment and can be traced to its continued popularity among the corporate enterprise software community. However, 15 years is a long time for any software framework to evolve, and it seems inevitable that it will pick up unwanted "baggage" along the way. Unfortunately, the .NET framework didn't escape this fate. In recent years, you could hear more and more grumbling in the community—things like:

- Too many OS-specific dependencies, preventing the framework from being cross-platform
- The framework has gotten too heavy, causing slow startup times
- .Net was too monolithic in the design and not modular enough
- Fragmented versions of .NET for different platforms
- Was not open source

To address these complaints, Microsoft decided it was time for a reboot, thus a project codenamed "Project K" was created in late 2014, the output of that project was what ultimately became .NET Core.

The New Microsoft

Before we talk about the specific features of .NET Core, we shouldn't overlook the fact this framework is created under the open source model (using MIT and Apache 2 licenses), with no ifs, ands, or buts. It's hosted on GitHub, free for anyone to examine, fork, make pull requests, or other contributions.

What's so great about .NET Core?

.NET Core represents the future of .NET, and it is where Microsoft is putting its energy moving forward (along with the community). The following are a few of the highlights of this "new" framework:

- Cross-platform: Runs on Windows, macOS and Linux.
- Flexible deployment: Can be included in your app or installed side-by-side, user-specific, or machine-wide
- Command-line tools: All product scenarios can be exercised at the command line
- Compatible: .NET Core is compatible with .NET Framework, Xamarin, and Mono, via the .NET Standard Library
- Fast: Numerous benchmarks show that it is on-par, if not faster, than other web stacks
- Supported by Microsoft: .NET Core is supported by Microsoft, per .NET Core Support
- A unified story for building web UI and web APIs
- Built-in dependency injection
- Cloud-ready, environment-based configuration system

- Container-friendly with its Linux support

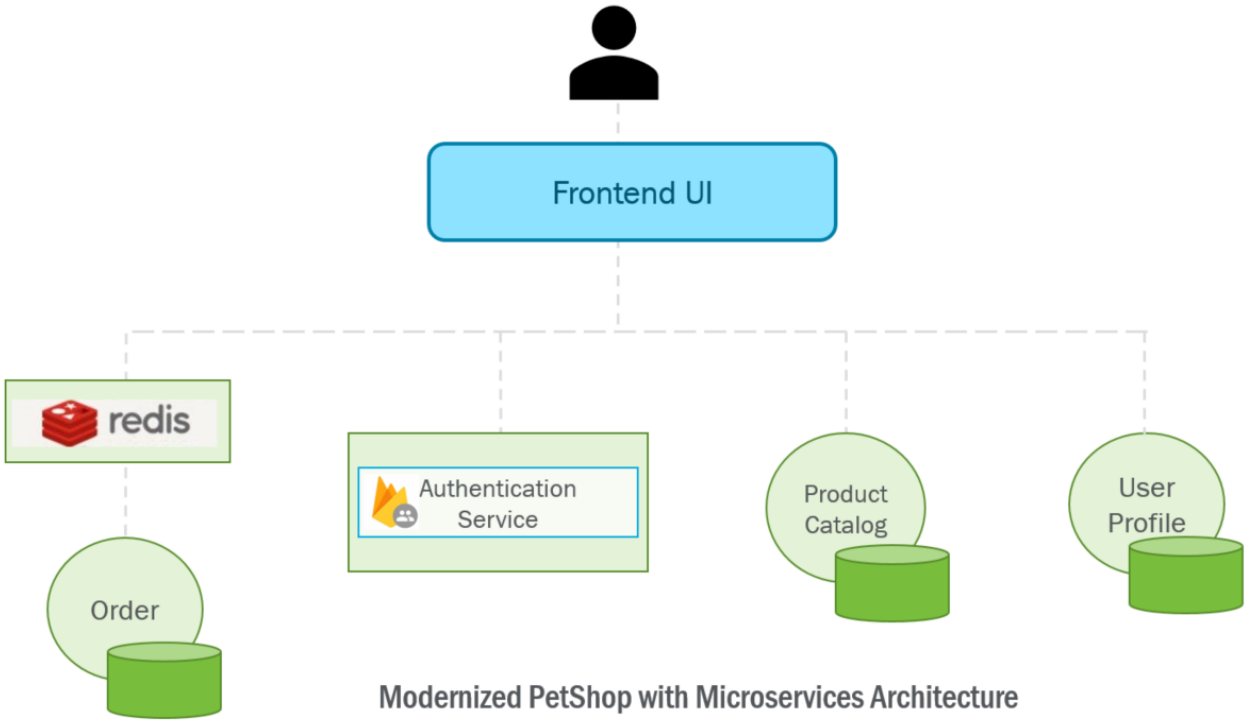
.NET Core is also lightweight and modular in its design, down to its core (no pun intended). This feature can be attested to by the fact the entire ASP.NET Core can be deployed via a NuGet package! This is not by accident—.NET Core was designed with modularity in mind. This modular design goal is propagated from the framework up, allowing applications to pull in specific features on an as-needed basis, which will provide tighter security (through smaller surface area), simplified app deployment, and improved performance. All of these features make this framework a great choice for developing cloud-friendly applications, namely, applications that have no host affinity and can start and stop at a moment's notice.

What are the implications for PetShop?

We made the determination early on that our actual UI Layer (i.e., the website developed with ASP.NET Web Forms), will remain intact for the most part (except for the changes needed to incorporate the Firebase Auth and business logic layers to call the new microservices). It will be migrated to the cloud as-is via lift-and-shift instead of being moved to .NET Core. As detailed in our first white paper in this series, these microservices are interface-compatible to how the PetShop's business logic layer expects the methods' signatures to be. As long as the implementations are call-compatible (in .NET interface terms), we are free to choose whatever technology we deem fit. In our case, ASP.NET Core would seem to be the logical fit (*Teaser alert: the choice for .NET Core will become even more apparent in the forthcoming final white paper of this series*). This is another one of the benefits of microservices architecture: call-compatibility for clients and services are only at the wire protocol and serialization level, not at the code implementation level. Hence there is no issue with the fact the client in this case is running in ASP.NET "Classic" Web Forms and our services are developed using ASP.NET Core (two technologies that were developed 12 years apart!). Microservices are, by definition, implementation-agnostic, and it is one of the main reason why so many developers and architects are embracing them.

Another reason for the choice of using ASP.NET Core to implement our microservices that may not be immediately apparent is that since .NET Core is cross-platform, it is much easier for us to use containers if we choose to do so; and that just happens to be the direction we are heading towards. The container use-case will be discussed at length in the final white paper of this series.

After all the application modernizations, the resulting architecture for PetShop will be as follow:



Modernized PetShop with Microservices Architecture

Conclusion

Let's do a bit of review before we conclude this white paper. What is the common theme among all the app modernization techniques that were utilized?

The answer is that they all have an open-source software licensing model. Is that by accident? Perhaps, but there is no question that the momentum behind OSS is gaining as it continues to provide some of the most advanced and updated technologies available today. To paraphrase Edward Wu, Director of Software Engineering at Niantic (maker of Pokemon Go), if you have developed something great, the best way to show it off is to openly share it with the rest of the world. Judging from the amount of great software contributions by the OSS community, there are lots of software developers that share a similar view. If nothing else, it's important for us, as software developers and architects, to continue to remind ourselves to not reinvent the wheel on basic system functionalities. Instead, try to harvest (or even make contribution to) the great software that is already available in the OSS community.

In keeping with our theme of adding value with each iteration, what we have achieved in this iteration is to breath new life into PetShop with updated technologies, SaaS, and other cloud-based managed service offerings. These improvements will offload a great deal of maintenance and management of these technologies to the provider of these services (Google Cloud Platform in this case), which in turn will enable us to focus on the core features and functionalities that will differentiate us from our competitors. As the industry moves forward, and takes more advantages of what cloud computing has to offer, this will be a repeating theme:

outsource, offload any features that's non-differentiated, shrink the surface area of the system to what's core, and leave the rest to the cloud providers.

What's next?

What constitutes a system being called cloud-native? This is somewhat of a controversial but important question as more companies are trying to squeeze every bit of competitive advantage out of their cloud investments. In our final white paper in this series, we will attempt to answer this key question and demonstrate what changes we needed to make in order to have our application fit this moniker. Stay tuned.