VSLive 2019

Demo–CloudMadeEasy-NET-Apps

Last updated: 8/12/2019

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Microsoft Contacts**   |  |  | | --- | --- | | Role | Individuals | | Owners | Michael Crump | | Speaker | Michael Crump | | Reviewers |  | |  |

## In this demo…

## With Visual Studio and Azure App Service, you can easily migrate your on-premise ASP.NET application to Azure

## With Durable Functions and C#, you can easily modernize your application with workflows to add functionality to your app without changing your existing codebase.

Demo flow

Take an on-premises ASP.NET Web Form app & SQL DB and migrate it to Azure; then modernize the app to support our new rewards program. We’ll show auto-scaling and use APIM to secure and monitor our API.

* Show the Customer entry web front-end –in Visual Studio (WebForms) and publish it to Azure
* Explain the benefits of Azure App Service – scaling, Azure Monitor (App Insights), Continuous Delivery, production debugging and staging slots.
* Add a Durable Function with 3 predefined workflows
* Add a customer mobile number inside of the Customer entry app and check the loyalty program box and it processes the workflow
* We’ll show the email being sent, and the other workflows which are “Declining” and “No Action Taken”

## Demo Set-up

Install the following items onto the demo machine(s):

1. Visual Studio 2019 with following workloads:
   1. ASP.NET
   2. .NET Core
   3. Azure
2. Azure Functions Tools for Visual Studio
3. [.NET Core 2.2 SDK](https://www.microsoft.com/net/download/dotnet-core/2.2)

Demo Walk-through

Take an on-premises ASP.NET Web Form app & SQL DB and migrate the ASP.NET app to Azure; then modernize the app to support our new rewards program. We’ll see a spike in traffic and show auto-scaling and APIM to secure and monitor our API in the future

|  |  |  |
| --- | --- | --- |
|  | 1. Screen opens to the internal web app hosted on-prem. | This web app is currently being used by customer support operators to answer questions about order status, etc. for orders that have already been placed.  This app originally implemented a loyalty program, but it relied on printed cards. To keep up with today’s retail challenges and customers on the move, they’d like to use the customers email address to send emails messages with special offers and discounts if they accept the terms of agreement.  But for now they’d like to include translation services for the customer’s native language |
| Maria | 1. Search for **Maria (or any user that is in the db)** in the client box, and press Enter to load that user. | This is just to show basic functionality of the web app by searching and finding a client by email address |
|  | 1. Switch over to show the **Tailwind.Traders.Rewards.Web** project structure in Visual Studio   Of note:  Demo details  \* Uses .NET Framework 3.5  \* ASP.NET WebForms  \* SQL Server DB | The web app consists of an **ASP.NET Web Forms** application we built some time ago, that talks directly to a **SQL Server** database.  We’re going to move the app along with the database to Azure |
| LOCALHOST | 1. Run it on localhost and provide a breakpoint on CustomerData.cs on line 87 2. Show that all the application does is toggles the Enrolled status. 3. Now we’re going to move it to Azure |  |
|  | 1. Walk through the **Publish** Dialog to deploy the web app. Click the **publish** button.   Walk through Publish   * Create new Profile * Add SQL Server * Edit existing profile and show DB connection | I’m going to deploy the ASP.NET app into Azure App Service using the simple publishing tools without leaving my favorite IDE - Visual Studio. |
|  | 1. Once **deployed**, click on the **link** provided in the **Visual Studio Web Publish Activity** to open the site in a browser. 2. Show the app is deployed to Azurewebsites.net | The ASP.NET application is a simple CRUD app, the kind of app you’ve probably built over and over again with ASP.NET Web Forms. From the URL, we can see that the web app is now live and no longer running locally. It is now hosted on Azure App Services. |
|  | 1. Switch over to the **Azure Portal** and go inside of the **App Service** to discuss several key benefits that you get by hosting **.NET apps in Azure**. | Benefits such as Deployment slots, Automatic backups, SSL, etc. |
| A screenshot of a computer screen  Description automatically generated | 1. Under the **Settings** option in the App Service menu, choose the **Scale Up** 2. A **Scale up** is the Azure websites cloud equivalent of moving your non-cloud web site to a bigger physical server. Scale up operations are useful to consider when your site is hitting a quota, signaling that you are outgrowing your existing mode or options. In addition, scaling up can be done on virtually any site without worrying about the implications of multi-instance data consistency. 3. Walk through configuring **Scale Out**. 4. **Scale out** operation is the equivalent of creating multiple copies of your web site and adding a load balancer to distribute the demand between them. When you scale out a web site in Windows Azure websites, there is no need to configure load balancing separately since this is already provided by the platform. | One of the challenges is managing the infrastructure and setting up the scalability needed for our apps. With App Service, we can scale up or down, manually or automatically. |
| CPU Rule | 1. Choose **Scale out** option from the blade and click **Enable auto-scale**. Name the rule as **Rewards scale** and select the Resource group if required. Add a rule that whenever the CPU percentage exceeds 70%, additional instances should be automatically created for the existing App service. Save the changes |  |
| Deployment Slots | 1. The **Deployment slots** which allows you to set up different environments for the application. Slots are useful for DevTest and production environments, A/B testing or probably to verify a core functionality before swapping the site to production. 2. Scroll down to the option **SSL Settings** where you can set up custom domains |  |
| App Insights | Another feature which is important is the ability to collect **logs and telemetry information** to monitor the application performance which is known as **Application Insights** in Azure. Click the feature and then click **Turn on site extension** to explore its services. | Also, in the Azure Portal, we can see a new Application Insight resource which will show the metrics of the app including Failures, Performance Availability, etc. We can also configure alerts, export options and Work Items creation. |
| Insights Data | 1. Take a look at your web application dashboard by clicking **View application insights data** to quickly notice the **server response time**, the **server requests**, **application exception failures**, etc. | Here, we can track all the metrics associated with our app, including failures and performance availability. We can also configure alerts, work items, and more. Let us review the exceptions. |
|  | 1. Introduce some errors by going to <http://ttrewardsrrzkxytcba4l6.azurewebsites.net/?yolo=hi> |  |
|  | 1. Go to Failures and exceptions and show the line number the error occurred on as well as show it in Visual Studio |  |
|  | 1. **Debug Snapshot** is another feature within the Application Insights that help you to download a snapshot of the exception to get an understanding of the state of the variables. You can load the downloaded snapshot into Visual Studio and engage the debugger to check the issues in the service that was running. It's almost like you're debugging in production except that you are bringing the production dump into your local Visual Studio and debug the exceptions locally. |  |
| Modernization Section | 1. Switch back to the **web browser** with our deployed site and call out the “**Enroll in loyalty program**” checkmark | We’d like to add on additional functionality to our app without changing our **production code** base. When they designed the website, they assumed they would have a loyalty program at some point but never got around to implementing it until now. What we’d like to do is check and see if a Mobile number is entered along with the “Enrolled in loyalty program” is checked and trigger a workflow, but how would we do that? |
|  | 1. Switch to the **Logic App** window in Edge and show the **SQL connector** that triggers based on updates to the SQL Table | We can quickly harness the power of Logic Apps to create a trigger on an SQL Server Table when our Customers table is modified (the “Enrolled in loyalty program” is checked.) Once an item is modified, it gets the row that was modified and creates a JSON object (with the fields shown in the image to the left) to our durable function. |
| Start the V2 Modernization that includes enrollment program |  |  |
|  | 1. Flip to the instance of Visual Studio that has the Azure Function opened | We decided to use Durable Functions since it lets you write stateful functions in a serverless environment and also manages state, checkpoints, and restarts for you.  We’ve created an Azure Function (durable) that   1. Invokes an activity to set that the customer is in the middle of a process and store its orchestration id 2. Sends a welcome email. 3. Waits for user acceptance, cancellation or timeout |
|  | 1. Review the Azure Function code that the Logic App triggers. It can be found in **EnrollmentHttpStart.cs**. | We’ll begin with the activity that checks to see if the JSON payload is 1) Already in the enrolment process or 2) Begin the Durable Function workflow to start the enrolment process. |
|  | 1. Review the Durable Function code that show cases the SMS responses for the different workflows. It can be found in **EnrollmentVerificator.cs** | We’ll begin with the code that 1) sends the initial “welcome” email if a customer is not enrolled in the program 2) review the code that triggers depending on if the user replies or clicks YES or NO and 3) Review the successful or cancelled messages that will be sent.  Optional: You may also review the code that triggers on a timeout that is found in **[FunctionName(Activities.SendReminderEnrollmentNotification)]** |
|  | 1. Back in Visual Studio, run the Function locally along with the ASP.NET website and show the APIs available in the Azure Function. | The **EnrollmentResponseNotification** sends URLs for the user accepting YES or NO  The **TwilioWebHook** is responsible for sending SMS notifications and accepts user input for YES OR NO  The **EnrollmentVerificatior\_HttpStart** starts the workflow |
|  | 1. Clear Storage Tables in local machine and talk about these tables |  |
|  | 1. Select the web app and type in **Maria** and check the “Enroll in loyalty program” and the checkbox will disappear, and you’ll see the enrolment is in progress | Run the ASP.NET website to show the site working with the local function |
|  | 1. Review the Azure Function host that shows the initial email is sent and awaits for a reply |  |
|  | 1. Switch to email and show the link has been sent |  |
|  | 1. Click accept and you’ll get another email saying you are now enrolled. |  |
|  | 1. Look at the other workflows for No and no response | EnrollmentVerificator line 44 and 137 |
|  | 1. Show publish button |  |
|  | 1. Show deployed function and show URL for API |  |
|  | 1. Show that the Logic app is using the same SQL connector but calling a POST method suppling the details. |  |