



Composite business apps in SharePoint 2013 and SharePoint Online solution pack

Microsoft Corporation

November 2014

**Applies to:** SharePoint 2013 and SharePoint Online

**Summary:** This solution pack includes code and documents that demonstrate and describe techniques for creating composite business apps in apps for SharePoint 2013 and SharePoint Online.

©2014 Microsoft Corporation. All rights reserved.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet website references, may change without notice. You bear the risk of using it.

Some examples are for illustration only and are fictitious. No real association is intended or inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes.

# Contents

This solution pack refers to apps that are tightly integrated with your business processes and other line of business technologies (databases, web services, etc.) as composite business apps. These apps generally need to encapsulate a number of relatively complex interactions, both with users and with other technologies, and they require some planning related to high level design.

The composite business apps solution pack focusses on the following building blocks that you may need to integrated into your apps and into your overall approach to the SharePoint 2013 app model:

* **Hosting approaches (SharePoint-hosted vs. provider-hosted)**
* **Migration of InfoPath forms capabilities to the app model**
* **Data storage models**
* Departmental apps that integrate with complex business processes. This solution pack uses an event planning app as a reference implementation
* Workflows with custom web service calls

This first module of the solution pack covers the hosting, InfoPath, and data storage building blocks. The second module discusses departmental apps, and the third module focusses on workflows.

Table 1 describes these three modules.

**Table 1. Composite business apps in SharePoint 2013 and SharePoint Online solution pack modules**

|  |  |  |
| --- | --- | --- |
| **Module** | **Name** | **Description** |
| **1** | **Hosting options, forms, and data** | **Describes appropriate use cases for SharePoint-hosted apps and provider-hosted apps. Provides guidance for moving InfoPath form capabilities to SharePoint 2013, and compares data storage options for the app model.** |
| 2 | Departmental apps | Describes a reference implementation of a corporate events app that demonstrates several ways to integrate apps for SharePoint into your business operations. |
| 3 | Workflows | Describes three code samples that demonstrate how to deploy a SharePoint 2013 workflow to the host web from an app for SharePoint and two approaches for calling web services from workflows. |

# SharePoint-hosted vs. provider-hosted apps

One of the first questions you’ll need to address when planning your business apps is where the app will be hosted. In general, SharePoint-hosted apps work best when you can easily scope your requirements to single site implementations that you can handle entirely with JavaScript. Provider-hosted apps tend to become more preferable as your business requirements become more complex.

Table 2 summarizes the factors that you should consider when deciding where to host your apps.

**Table 2. SharePoint-hosted apps vs. provider-hosted apps**

|  |  |
| --- | --- |
| **SharePoint-hosted apps** | **Provider-hosted apps** |
| You can do everything you need to do with JavaScript. | You need to use languages other than JavaScript. |
| The app does not need to do any work across more than one site: team calendar apps and featured news rotators, for example. | The app needs to access information and do work across more than a single site: site collection provisioning apps, for example. |
| Content is sensitive and needs to stay securely and entirely in SharePoint. | The app needs to integrate with other line of business technologies. |
|  | The app requires elevated permissions (possible with the app-only policy). |
|  | The app requires a highly customized user interface. |

# Migrating InfoPath forms capabilities to SharePoint 2013

**The essentials**. InfoPath 2013 is the last release of the desktop InfoPath client, and InfoPath Forms Services in SharePoint Server 2013 is the last release of InfoPath Forms Services. The client and the on-premises version of InfoPath Forms Services in SharePoint Server 2013 will be fully supported until 2023. The forms service will be supported on Office 365 until at least the next major release of Office. Now is a good time to start migrating your forms capabilities to other supported approaches. This section describes those other approaches and when you’ll want to use them.

**The alternatives.** These are the alternatives in SharePoint 2013 to capabilities that you’re currently getting from InfoPath:

* Access applications
* Convert InfoPath forms to Sandbox solutions
* Move complex behaviors to the app model

You should strongly consider the first two, since they can be implemented by information workers who don’t necessarily know how to write and deploy code-based alternatives. Table 3 describes the scenarios I which each alternative will work best.

**Table 3. InfoPath alternatives in SharePoint 2013**

|  |  |
| --- | --- |
| **Alternative** | **Scenario** |
| Access applications | This option supports multiple forms that handle relational data contained in multiple Access tables, Excel tables, and/or SharePoint lists. |
| Move InfoPath forms to Sandbox | This is a viable option if your forms are simple, don’t require any server components, and don’t make any calls to external web services. |
| Convert to the app model | You can convert complex forms driven by extensive code into provider-hosted apps. This option requires developer resources. |

# Data storage models and options

This section describes a code sample that demonstrates six data storage options that you can use with apps for SharePoint:

* SharePoint list on the host web
* SharePoint list on the app web
* SQL Azure database
* Azure queue storage
* Azure table storage
* External web service

The sample applies each data storage option to a specific function for which it’s well suited. The app implements a customer service dashboard and related interfaces that display recent orders, customer representative survey ratings, customer notes, support cases, and a customer representative call queue. Figure 1 shows which option applies to each function.

**Figure 1. Data storage options tied to each function in the data storage sample**



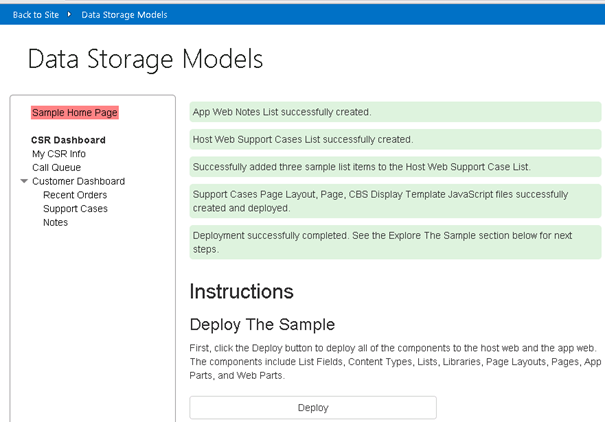
## [Data storage models for apps](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.DataStorageModels)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| This provider-hosted sample application for SharePoint demonstrates the differences, advantages, and disadvantages between different data storage patterns associated with the App Model and how they are built. | This sample illustrates the strengths and limitations associated with each kind of data storage component that you should consider when deciding which data storage models to use when building apps for SharePoint. More generally, it demonstrates the advantages of remotely storing the data used in your apps. | This sample app is a provider-hosted app written in C# and JavaScript that deploys a number of SharePoint artifacts (lists, app part, web part, etc.) to both the host web and the app web. It interacts with both SharePoint lists on the app web and host web, and also makes calls to a SQL Azure database, Azure queue and table storage, and a remote web service that implements OData. It uses the Model-View-Controller (MVC) pattern. |

This app requires you to have a Microsoft Azure account where you can deploy a SQL Azure database and create an Azure storage account. You should also have a SharePoint developer site so that you can deploy the sample from Visual Studio 2013. See the [Core.StorageDataModels](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.DataStorageModels) project for complete instructions for deploying the sample and the Azure components that it needs.

The app’s interface prompts you to deploy the SharePoint components that the app uses (Figure 2). The left column also contains links to the six scenarios covered in the sample (a CSR dashboard showing customer representative survey ratings, customer call queue, and a dashboard showing recent orders, support cases, and customer notes).

**Figure 2. Data storage models launch page prompts you to deploy SharePoint components**

[](https://camo.githubusercontent.com/d4bb65a81eafc9c90583ef8b48261b1151e0ee52/687474703a2f2f692e696d6775722e636f6d2f637253343037512e706e67)

We’ll start by discussing the two examples that are supported by storing data in SharePoint lists: the support cases and customer notes scenarios. They both share one significant advantage: you can easily retrieve the data with relatively simple client object model code or REST queries. They also share one big disadvantage: list query thresholds on both the app web and the host web.

## Notes scenario (SharePoint list on the app web)

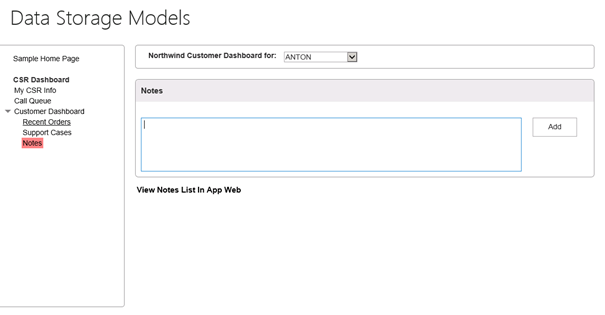
**Advantage**: You can easily query the data with relatively simple REST queries or client object model code.

**Disadvantages**:

|  |  |  |
| --- | --- | --- |
| **Design** | **Performance** | **Backup/Restore** |
| * You can’t update the list metadata without updating and redeploying the app. * If you update the data structure, you’ll have to rewrite your application logic for storing and updating the information. * You can’t easily use the information stored in the list with other apps and applications. * You can’t search for the data in SharePoint * The app web limits both the amount of data you can store in lists and the size of query results. If your business needs require storing and/or querying large data sets, this isn’t a good alternative. | Databases generally provide better performance for complex queries.  Databases support complex data structures more elegantly. | * Databases generally provide better options for backing up and restoring data. * Lists on the app web are deleted whenever an app is uninstalled and you lose all of the data stored in them, unless the app uses the **AppUninstalled** event to copy and store the data somewhere else. |

The code underlying the **Notes** section of the customer dashboard uses REST queries to retrieve data from a list that the app has deployed to the app web. This list contains fields for titles, authors, customer IDs, and descriptions. The interface allows you to add and retrieve notes for a customer that you specify (Figure 3).

**Figure 3. Notes interface**

[](https://camo.githubusercontent.com/440879b750fa347a10bef06fd2490a16ca90e693/687474703a2f2f692e696d6775722e636f6d2f705842575149312e706e67)

The **View Notes List in App Web** link allows you to see an “out of the box” view of the list data.

This app uses the Model-View-Controller (MVC) pattern. You can see the code for the notes scenario in the Views/CustomerDashboard/Notes.cshtml file. It uses simple REST calls to add and retrieve data.

The following code retrieves notes for a given customer from the notes list.

function getNotesAndShow() {

var executor = new SP.RequestExecutor(appWebUrl);

executor.executeAsync(

{

url: appWebUrl + "/\_api/web/lists/getByTitle('Notes')/items/" +

"?$select=FTCAM\_Description,Modified,Title,Author/ID,Author/Title" +

"&$expand=Author/ID,Author/Title" +

"&$filter=(Title eq '" + customerID + "')",

type: "GET",

dataType: 'json',

headers: { "accept": "application/json;odata=verbose" },

success: function (data) {

var value = JSON.parse(data.body);

showNotes(value.d.results);

},

error: function (error) { console.log(JSON.stringify(error)) }

}

);

}

The following code adds a note for a given customer to the notes list.

function addNoteToList(note, customerID) {

var executor = new SP.RequestExecutor(appWebUrl);

var bodyProps = {

'\_\_metadata': { 'type': 'SP.Data.NotesListItem' },

'Title': customerID,

'FTCAM\_Description': note

};

executor.executeAsync({

url: appWebUrl + "/\_api/SP.AppContextSite(@target)/web/lists/getbytitle('Notes')/items?@target='" + appWebUrl + "'",

contentType: "application/json;odata=verbose",

method: "POST",

headers: {

"accept": "application/json;odata=verbose",

"content-type": "application/json;odata=verbose",

"X-RequestDigest": $("#\_\_REQUESTDIGEST").val()

},

body: JSON.stringify(bodyProps),

success: getNotesAndShow,

error: addNoteFailed

});

}

The [Core.DataStorageModels](https://github.com/OfficeDev/PnP/tree/master/Samples/Core.DataStorageModels) project on GitHub contains instructions for adding 5000 items to the list to demonstrate that list queries that generate a result set of 5000 or more items will hit the list query threshold and fail. It also contains instructions for adding so much data to your list on the app web that it causes you to exceed the storage limit for your site collection (which depends on how much storage space you’ve allocated to it). These pieces of the app demonstrate two of the most important limitations of this approach: list query size limits and storage space limits. If your business needs require you to work with large data sets and query result sets, this approach won’t work.

## 4.3 Support cases scenario (SharePoint list on the host web)

**Advantages**:

* You can easily query the data with relatively simple REST queries or client object model code.
* You can search for the data in SharePoint.
* You can update the list metadata and create new views for a list without updating and redeploying the app. These changes won’t affect the behavior of your app.
* Lists on the host web are not deleted whenever an app is uninstalled, unless the app uses the **AppUninstalled** event to remove the list and/or delete the data.

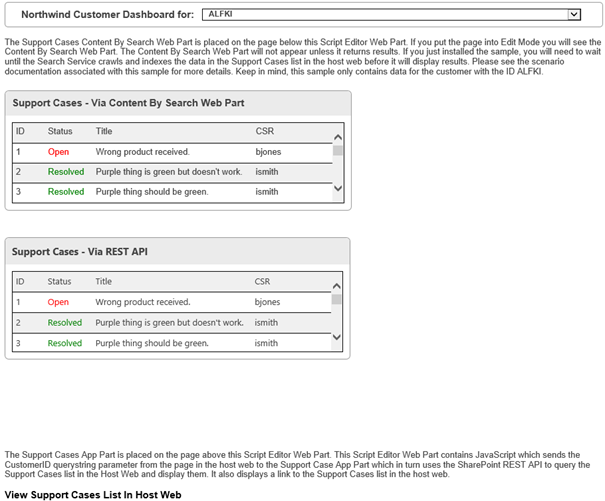
**Disadvantages**:

|  |  |  |
| --- | --- | --- |
| **Design** | **Performance** | **Backup/Restore** |
| The host web limits both the amount of data you can store in lists and the size of query results. If your business needs require storing and/or querying large data sets, this isn’t a good alternative. | Databases generally provide better performance for complex queries. | Databases generally provide better options for backing up and restoring data. |

The support cases scenario stores its data in a SharePoint list deployed to the host web, and it retrieves and displays data from this list in two ways: a [Content Search Web Part](http://msdn.microsoft.com/en-us/library/office/jj163789(v=office.15).aspx) (CSWP) and an app part that’s implemented as an MVC view. The code in this view uses REST queries to retrieve information from the list, while the CSWP uses SharePoint’s search service to retrieve the data. The two approaches demonstrate a significant advantage of this alternative: you can use both the search service and the REST/CSOM APIs to retrieve information from a list on the host web.

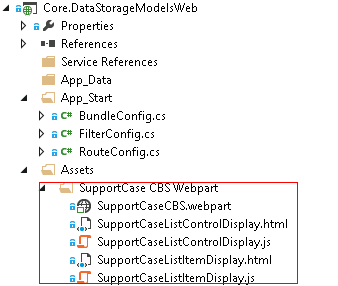
When you select a customer from the support cases drop-down, you’ll see the support case data for that customer displayed in both the CSWP and the app part (Figure 4). The CSWP might not return content right away, since it can take up to 24 hours for the SharePoint search service to index the data. As in the Notes scenario, you can also click on the **View Support Cases List in Host Web** link to see a conventional view of the list data.

**Figure 4. Support cases interface**

[](https://camo.githubusercontent.com/8477f67bd0793dd2b05ae91f3a74a40cbf6d7097/687474703a2f2f692e696d6775722e636f6d2f67625a78796f682e706e67)

The CSWP deployed by this app uses a custom display template. Figure 5 shows where in the **Assets** directory of the web project you can find the web part and the associated template.

**Figure 5. Content search web part and display**



The following JavaScript in the Views/SupportCaseAppPart\Index.cshtml file uses the cross-domain library to make a REST query to the SharePoint list on the host web.

function execCrossDomainRequest() {

var executor = new SP.RequestExecutor(appWebUrl);

executor.executeAsync(

{

url: appWebUrl + "/\_api/SP.AppContextSite(@@target)" +

"/web/lists/getbytitle('Support Cases')/items" +

"?$filter=(FTCAM\_CustomerID eq '" + customerID + "')" +

"&$top=30" +

"&$select=Id,Title,FTCAM\_Status,FTCAM\_CSR" +

"&@@target='" + hostWebUrl + "'",

method: "GET",

headers: { "Accept": "application/json; odata=verbose" },

success: successHandler,

error: errorHandler

}

);

}

The [Core.DataStorageModels](https://github.com/OfficeDev/PnP/tree/master/Samples/Core.DataStorageModels) project on GitHub contains instructions for adding 5000 items to the list to demonstrate that list queries that generate a result set of 5000 or more items will hit the list query threshold and fail. This piece of the app demonstrates one of the most important limitations of this approach: list query size limits. If your business needs require you to work with large data and query result sets, this approach won’t work.

The other four scenarios uses different kinds of remote storage. The first one uses a web service that returns JSON-formatted customer information.

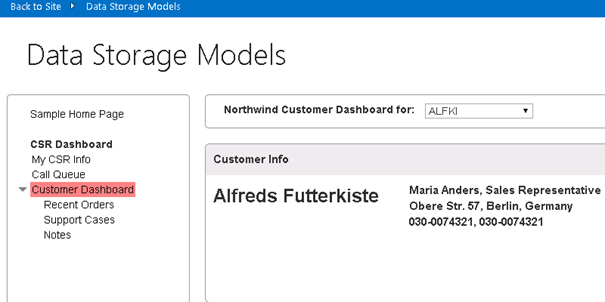
## 4.4 Customer dashboard scenario (Northwind OData web service)

**Advantages**:

|  |  |  |
| --- | --- | --- |
| **Design** | **Performance** | **Backup/Restore** |
| * Your web service can support more than one app. * You can make updates to your web service without having to update and redeploy your app. | * Your SharePoint installation and web service have no impact on each other’s performance. * Hosting services such as Microsoft Azure provide easy ways to scale your web services. | * You can back up and restore information on your web services separately from your SharePoint infrastructure. * You don’t lose any data when you uninstall your app, unless the app uses the **AppUninstalled** event to delete the data. |

The customer dashboard scenario stores its data in a web service that implements the [OData standard](http://www.odata.org/) for retrieving data. Figure 6 shows how the customer dashboard interface works. You select a customer from a drop-down menu, and customer information displays in the customer info pane.

**Figure 6. Customer dashboard scenario**

[](https://camo.githubusercontent.com/f5eefe5bb77c342202cd480fef78b5b3fde1760e/687474703a2f2f692e696d6775722e636f6d2f6343523977306f2e706e67)

This page is an MVC view whose display is defined in the Views/CustomerDashboard\Home.cshtml file. The underlying code is in the Scripts/CustomerDashboard.js file. The JavaScript code uses AJAX to query the Northwind web service. Since this is an OData service, the web service query consists entirely of query string arguments attached to a URL that points to a web service endpoint. The service returns customer information in JSON format.

The following code runs when you select the **Customer Dashboard** link. It retrieves all of the customer names and IDs in order to populate the drop-down menu.

var getCustomerIDsUrl = "https://odatasampleservices.azurewebsites.net/V3/Northwind/Northwind.svc/Customers?$format=json&$select=CustomerID";

$.get(getCustomerIDsUrl).done(getCustomerIDsDone)

.error(function (jqXHR, textStatus, errorThrown) {

$('#topErrorMessage').text('Can\'t get customers. An error occurred: ' + jqXHR.statusText);

});

The following code runs when you select a customer name from the drop-down menu. It uses the OData **$filter** argument to specify the customer ID and other query string arguments to retrieve information related to this customer.

var url = "https://odatasampleservices.azurewebsites.net/V3/Northwind/Northwind.svc/Customers?$format=json" + "&$select=CustomerID,CompanyName,ContactName,ContactTitle,Address,City,Country,Phone,Fax" + "&$filter=CustomerID eq '" + customerID + "'";

$.get(url).done(getCustomersDone)

.error(function (jqXHR, textStatus, errorThrown) {

alert('Can\'t get customer ' + customerID + '. An error occurred: ' +

jqXHR.statusText);

});

The last three scenarios use different types of remote storage that are provided by the Microsoft Azure hosting service.

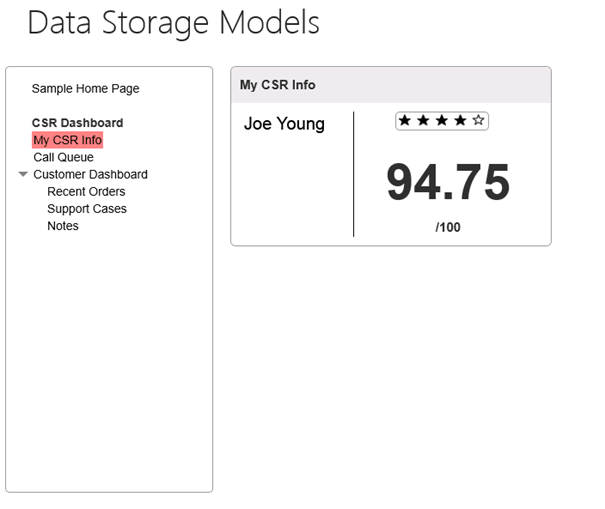
## 4.5 Customer service representative survey scenario (Azure Table Storage)

**Advantages**:

|  |  |  |
| --- | --- | --- |
| **Design** | **Performance** | **Backup/Restore** |
| * Azure Storage tables can support more than one app. * You can make updates to the schemas of your Azure Storage tables without having to update and redeploy your app. | * Your SharePoint installation and Azure Storage tables have no impact on each other’s performance. * Azure Storage tables scale easily. | * You can back up and restore information on your Azure Storage tables separately from your SharePoint infrastructure. * You don’t lose any data when you uninstall your app, unless the app uses the **AppUninstalled** event to delete the data. |

The app’s interface displays the current user’s survey rating in the center page (Figure 7). If that Azure Storage table is empty, the app will add some information to the table before displaying it.

**Figure 7. Customer representative survey scenario**

[](https://camo.githubusercontent.com/ceac8eef4c93a3cf9025c7a2a91b36eda2678eb9/687474703a2f2f692e696d6775722e636f6d2f32754e5146464c2e706e67)

This page is an MVC view that is defined in the Views\CSRInfo\Home.cshmtl file. The Controllers\CSRInfoController.cs class contains a **Home** method that uses the SharePoint client object model to retrieve the current SharePoint user **nameId**. It then passes the **nameId** value to the **GetUserScore** method that is defined in the Services\SurveyRatingService.cs file.

The following code from the CSRInfoController.cs defines the **Home** method that retrieves the user’s **nameId**.

[SharePointContextFilter]

public ActionResult Home()

{

var context =

SharePointContextProvider.Current.GetSharePointContext(HttpContext);

var sharePointService = new SharePointService(context);

var currentUser = sharePointService.GetCurrentUser();

ViewBag.UserName = currentUser.Title;

var surveyRatingsService = new SurveyRatingsService();

ViewBag.Score = surveyRatingsService.GetUserScore(currentUser.UserId.NameId);

return View();

}

The following code from the SurveyRatingService.cs file defines the **SurveyRatingsService** constructor, which sets up the connection to the Azure Storage account.

public SurveyRatingsService(string storageConnectionStringConfigName =

"StorageConnectionString")

{

var connectionString = Util.GetConfigSetting("StorageConnectionString");

var storageAccount = CloudStorageAccount.Parse(connectionString);

this.tableClient = storageAccount.CreateCloudTableClient();

this.surveyRatingsTable = this.tableClient.GetTableReference("SurveyRatings");

this.surveyRatingsTable.CreateIfNotExists();

}

The following code from the same file defines the **GetUserScore** method, which retrieves the user’s survey score from the Azure Storage table.

public float GetUserScore(string userName)

{

var query = new TableQuery<Models.Customer>()

.Select(new List<string> { "Score" })

.Where(TableQuery.GenerateFilterCondition("Name",

QueryComparisons.Equal, userName));

var items = surveyRatingsTable

.ExecuteQuery(query)

.ToArray();

if (items.Length == 0)

return AddSurveyRatings(userName);

return (float)items.Average(c => c.Score);

}

If the table doesn’t contain any survey data related to the current user, the AddSurveyRating method will randomly assign a score for the user.

private float AddSurveyRatings(string userName)

{

float sum = 0;

int count = 4;

var random = new Random();

for (int i = 0; i < count; i++)

{

var score = random.Next(80, 100);

var customer = new Models.Customer(Guid.NewGuid(), userName, score);

var insertOperation = TableOperation.Insert(customer);

surveyRatingsTable.Execute(insertOperation);

sum += score;

}

return sum / count;

}

## 4.6 Customer call queue scenario (Azure Queue Storage)

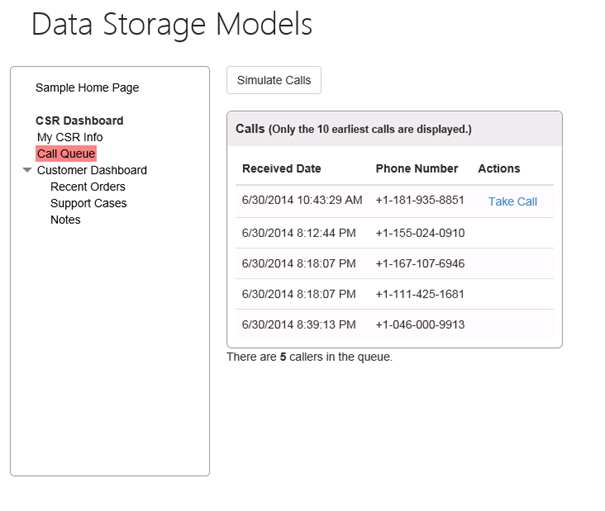
**Advantages**:

|  |  |  |
| --- | --- | --- |
| **Design** | **Performance** | **Backup/Restore** |
| * Azure Storage queues can support more than one app. * You can make updates to your Azure Storage queues without having to update and redeploy your app. | * Your SharePoint installation and Azure Storage queues have no impact on each other’s performance. * Azure Storage queues scale easily. | * You can back up and restore information on your Azure Storage queues separately from your SharePoint infrastructure. * You don’t lose any data when you uninstall your app, unless the app uses the **AppUninstalled** event to delete the data. |

The Call Queue scenario lists callers in a customer service support queue and simulates taking calls from customers. The call queue scenario uses an Azure Storage queue for data storage and the **Microsoft.WindowsAzure.Storage.Queue.CloudQueue** API to retrieve data from the queue.

The app’s interface displays a support call queue in the center pane when you click on the Call Queue link (Figure 8). You can simulate receiving calls (adding a call to the queue) by clicking the **Simulate Calls** button, and you can simulate taking the oldest call (removing a call from the queue) by clicking the **Take Call** action associated with a given call.

**Figure 8. Call queue scenario**

[](https://camo.githubusercontent.com/e48d168bd7e6e3dee4933741ba8c4dd019198a22/687474703a2f2f692e696d6775722e636f6d2f6867674e74486d2e706e67)

This page is an MVC view that is defined in the Views\CallQueue\Home.cshmtl file. The Controllers\CallQueueController.cs file defines the CallQueueController class, which contains methods for retrieving all calls in the queue, adding a call to the queue (simulating a call), and removing a call from the queue (taking a call). Each of these methods calls methods defined in the Services\CallQueueService.cs file, which uses the Azure Storage Queue API to retrieves the underlying information in the storage queue.

public class CallQueueController : Controller

{

public CallQueueService CallQueueService { get; private set; }

public CallQueueController()

{

CallQueueService = new CallQueueService();

}

// GET: CallQueue

public ActionResult Home(UInt16 displayCount = 10)

{

var calls = CallQueueService.PeekCalls(displayCount);

ViewBag.DisplayCount = displayCount;

ViewBag.TotalCallCount = CallQueueService.GetCallCount();

return View(calls);

}

[HttpPost]

public ActionResult SimulateCalls(string spHostUrl)

{

int count = CallQueueService.SimulateCalls();

TempData["Message"] = string.Format("Successfully simulated {0} calls and added them to the call queue.", count);

return RedirectToAction("Index", new { SPHostUrl = spHostUrl });

}

[HttpPost]

public ActionResult TakeCall(string spHostUrl)

{

CallQueueService.DequeueCall();

TempData["Message"] = "Call taken successfully and removed from the call queue!";

return RedirectToAction("Index", new { SPHostUrl = spHostUrl });

}

}

The CallQueueService.cs file defines the **CallQueueService** class, which establishes the connection to the Azure Storage queue. That class also contains the methods for adding, dequeuing (removing), and retrieving the calls from the queue.

public class CallQueueService

{

private CloudQueueClient queueClient;

private CloudQueue queue;

public CallQueueService(string storageConnectionStringConfigName = "StorageConnectionString")

{

var connectionString = CloudConfigurationManager.GetSetting(storageConnectionStringConfigName);

var storageAccount = CloudStorageAccount.Parse(connectionString);

this.queueClient = storageAccount.CreateCloudQueueClient();

this.queue = queueClient.GetQueueReference("calls");

this.queue.CreateIfNotExists();

}

public int? GetCallCount()

{

queue.FetchAttributes();

return queue.ApproximateMessageCount;

}

public IEnumerable<Call> PeekCalls(UInt16 count)

{

var messages = queue.PeekMessages(count);

var serializer = new JavaScriptSerializer();

foreach (var message in messages)

{

Call call = null;

try

{

call = serializer.Deserialize<Call>(message.AsString);

}

catch { }

if (call != null) yield return call;

}

}

public void AddCall(Call call)

{

var serializer = new JavaScriptSerializer();

var content = serializer.Serialize(call);

var message = new CloudQueueMessage(content);

queue.AddMessage(message);

}

public void DequeueCall()

{

var message = queue.GetMessage();

queue.DeleteMessage(message);

}

public int SimulateCalls()

{

Random random = new Random();

int count = random.Next(1, 6);

for (int i = 0; i < count; i++)

{

int phoneNumber = random.Next();

var call = new Call

{

ReceivedDate = DateTime.Now,

PhoneNumber = phoneNumber.ToString("+1-000-000-0000")

};

AddCall(call);

return count;

}

}

## 4.7 Recent orders scenario (SQL Azure database)

**Advantages**:

|  |  |  |
| --- | --- | --- |
| **Design** | **Performance** | **Backup/Restore** |
| * A database can support more than one app. * You can make updates to your database schema without having to update and redeploy your app, as long as the schema changes don’t affect the queries in your app. * A relational database can support many-to-many relationships and thus support more complex business scenarios. * You can use database design tools to optimize the design of your database. | Relational databases provide better performance than the other options when you need to execute complex operations in your queries, such as calculations and joins. | * A SQL Azure database allows you to import and export data more easily, so it’s easier to manage and move your data. * You don’t lose any data when you uninstall your app, unless the app uses the **AppUninstalled** event to delete the data. |

The recent orders interface works much like the customer dashboard interface. You click on the **Recent Orders** link in the left column, and then choose a customer from the drop-down menu at the top of the center pane. A list of orders from that customer will appear in the center pane.

This page is an MVC view defined in the Views\CustomerDashboard\Orders.cshtml file. Code in the Controllers\CustomerDashboardController.cs file uses the Entity Framework to query the Orders table in your SQL Azure database. The customer ID is passed via a query string parameter in the URL that is passed when the user selects a customer from the drop-down menu. The query joins the Customer, Employee and Shipper tables. The query result is then passed to the MVC view that displays the results.

The following code from the CustomerDashboardController.cs file performs the database query and returns the data to the view.

public ActionResult Orders(string customerId)

{

Order[] orders;

using (var db = new NorthWindEntities())

{

orders = db.Orders

.Include(o => o.Customer)

.Include(o => o.Employee)

.Include(o => o.Shipper)

.Where(c => c.CustomerID == customerId)

.ToArray();

}

ViewBag.SharePointContext =

SharePointContextProvider.Current.GetSharePointContext(HttpContext);

return View(orders);

}