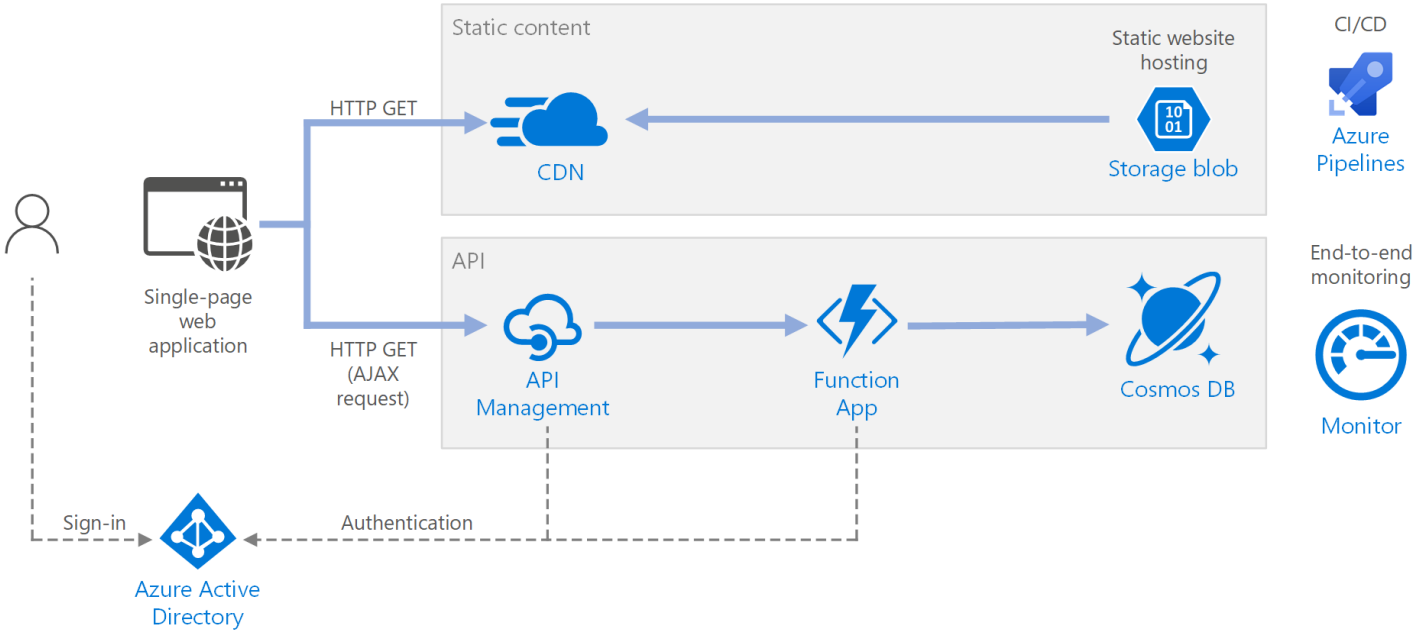
Serverless web application on Azure

05/28/2019 • 15 minutes to read • Contributors

In this article

This reference architecture shows a serverless web application. The application serves static content from Azure Blob Storage, and implements an API using Azure Functions. The API reads data from Cosmos DB and returns the results to the web app.

A reference implementation for this architecture is available on GitHub.



The term serverless has two distinct but related meanings:

- **Backend as a service** (BaaS). Backend cloud services, such as databases and storage, provide APIs that enable client applications to connect directly to these services.
- **Functions as a service** (FaaS). In this model, a "function" is a piece of code that is deployed to the cloud and runs inside a hosting environment that completely abstracts the servers that run the code.

Both definitions have in common the idea that developers and DevOps personnel don't need to deploy, configure, or manage servers. This reference architecture focuses on FaaS using Azure Functions, although serving web content from Azure Blob Storage is an example of BaaS. Some important characteristics of FaaS are:

1. Compute resources are allocated dynamically as needed by the platform.
2. Consumption-based pricing: You are charged only for the compute resources used to execute your code.
3. The compute resources scale on demand based on traffic, without the developer needing to do any configuration.

Functions are executed when an external trigger occurs, such as an HTTP request or a message arriving on a queue. This makes an event-driven architecture style natural for serverless architectures. To coordinate work between components in the architecture, consider using message brokers or pub/sub patterns. For help choosing between messaging technologies in Azure, see Choose between Azure services that deliver messages.

Architecture

The architecture consists of the following components:

Blob Storage. Static web content, such as HTML, CSS, and JavaScript files, are stored in Azure Blob Storage and served to clients by using static website hosting. All dynamic interaction happens through JavaScript code making calls to the backend APIs. There is no server-side code to render the web page. Static website hosting supports index documents and custom 404 error pages.

CDN. Use Azure Content Delivery Network (CDN) to cache content for lower latency and faster delivery of content, as well as providing an HTTPS endpoint.

Function Apps. Azure Functions is a serverless compute option. It uses an event-driven model, where a piece of code (a "function") is invoked by a trigger. In this architecture, the function is invoked when a client makes an HTTP request. The request is always routed through an API gateway, described below.

API Management. API Management provides a API gateway that sits in front of the HTTP function. You can use API Management to publish and manage APIs used by client applications. Using a gateway helps to decouple the front-end application from the back-end APIs. For example, API Management can rewrite URLs, transform requests before they reach the backend, set request or response headers, and so forth.

API Management can also be used to implement cross-cutting concerns such as:

- Enforcing usage quotas and rate limits
- Validating OAuth tokens for authentication
- Enabling cross-origin requests (CORS)
- Caching responses
- Monitoring and logging requests

If you don't need all of the functionality provided by API Management, another option is to use Functions Proxies. This feature of Azure Functions lets you define a single API surface for multiple function apps, by creating routes to back-end functions. Function proxies can also perform limited transformations on the HTTP request and response. However, they don't provide the same rich policy-based capabilities of API Management.

Cosmos DB. Cosmos DB is a multi-model database service. For this scenario, the function application fetches documents from Cosmos DB in response to HTTP GET requests from the client.

Azure Active Directory (Azure AD). Users sign into the web application by using their Azure AD credentials. Azure AD returns an access token for the API, which the web application uses to authenticate API requests (see Authentication).

Azure Monitor. Monitor collects performance metrics about the Azure services deployed in the solution. By visualizing these in a dashboard, you can get visibility into the health of the solution. It also collected application logs.

Azure Pipelines. Pipelines is a continuous integration (CI) and continuous delivery (CD) service that builds, tests, and deploys the application.

Recommendations

Function App plans

Azure Functions supports two hosting models. With the **consumption plan**, compute power is automatically allocated when your code is running. With the **App Service** plan, a set of VMs are allocated for your code. The App Service plan defines the number of VMs and the VM size.

Note that the App Service plan is not strictly *serverless*, according to the definition given above. The programming model is the same, however — the same function code can run in both a consumption plan and an App Service plan.

Here are some factors to consider when choosing which type of plan to use:

- **Cold start**. With the consumption plan, a function that hasn't been invoked recently will incur some additional latency the next time it runs. This additional latency is due to allocating and preparing the runtime environment. It is usually on the order of seconds but depends on several factors, including the number of dependencies that need to be loaded. For more information, see Understanding Serverless Cold Start. Cold start is usually more of a concern for interactive workloads (HTTP triggers) than asynchronous message-driven workloads (queue or event hubs triggers), because the additional latency is directly observed by users.
- **Timeout period**. In the consumption plan, a function execution times out after a configurable period of time (to a maximum of 10 minutes)
- **Virtual network isolation**. Using an App Service plan allows functions to run inside of an App Service Environment, which is a dedicated and isolated hosting environment.
- **Pricing model**. The consumption plan is billed by the number of executions and resource consumption (memory × execution time). The App Service plan is billed hourly based on VM instance SKU. Often, the consumption plan can be cheaper than an App Service plan, because you pay only for the compute resources that you use. This is especially true if your traffic experiences peaks and troughs. However, if an application experiences constant high-volume throughput, an App Service plan may cost less than the consumption plan.
- **Scaling**. A big advantage of the consumption model is that it scales dynamically as needed, based on the incoming traffic. While this scaling occurs quickly, there is still a ramp-up period. For some workloads, you might want to deliberately overprovision the VMs, so that you can handle bursts of traffic with zero ramp-up time. In that case, consider an App Service plan.

Function App boundaries

A *function app* hosts the execution of one or more *functions*. You can use a function app to group several functions together as a logical unit. Within a function app, the functions share the same application settings, hosting plan, and deployment lifecycle. Each function app has its own hostname.

Use function apps to group functions that share the same lifecycle and settings. Functions that don't share the same lifecycle should be hosted in different function apps.

Consider taking a microservices approach, where each function app represents one microservice, possibly consisting of several related functions. In a microservices architecture, services should have loose coupling and high functional cohesion. *Loosely* coupled means you can change one service without requiring other services to be updated at the same time. *Cohesive* means a service has a single, well-defined purpose. For more discussion of these ideas, see Designing microservices: Domain analysis.

Function bindings

Use Functions bindings when possible. Bindings provide a declarative way to connect your code to data and integrate with other Azure services. An input binding populates an input parameter from an external data source. An output binding sends the function's return value to a data sink, such as a queue or database.

For example, the `GetStatus` function in the reference implementation uses the Cosmos DB input binding. This binding is configured to look up a document in Cosmos DB, using query parameters that are taken from the query string in the HTTP request. If the document is found, it is passed to the function as a parameter.

```
[FunctionName("GetStatusFunction")]
public static Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", Route = null)] HttpRequest req,
    [CosmosDB(
        databaseName: "%COSMOSDB_DATABASE_NAME%",
        collectionName: "%COSMOSDB_DATABASE_COL%",
        ConnectionStringSetting = "COSMOSDB_CONNECTION_STRING",
        Id = "{Query.deviceId}",
        PartitionKey = "{Query.deviceId}")] dynamic deviceStatus,
    ILogger log)
{
    ...
}
```

By using bindings, you don't need to write code that talks directly to the service, which makes the function code simpler and also abstracts the details of the data source or sink. In some cases, however, you may need more complex logic than the binding provides. In that case, use the Azure client SDKs directly.

Scalability considerations

Functions. For the consumption plan, the HTTP trigger scales based on the traffic. There is a limit to the number of concurrent function instances, but each instance can process more than one request at a time. For an App Service plan, the HTTP trigger scales according to the number of VM instances, which can be a fixed value or can autoscale based on a set of autoscaling rules. For information, see Azure Functions scale and hosting.

Cosmos DB. Throughput capacity for Cosmos DB is measured in Request Units (RU). A 1-RU throughput corresponds to the throughput need to GET a 1KB document. In order to scale a Cosmos DB container past 10,000 RU, you must specify a partition key when you create the container and include the partition key in every document that you create. For more information about partition keys, see Partition and scale in Azure Cosmos DB.

API Management. API Management can scale out and supports rule-based autoscaling. Note that the scaling process takes at least 20 minutes. If your traffic is bursty, you should provision for the maximum burst traffic that you expect. However, autoscaling is useful for handling hourly or daily variations in traffic. For more information, see Automatically scale an Azure API Management instance.

Disaster recovery considerations

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of the geo-distribution features in the various services:

- API Management supports multi-region deployment, which can be used to distribute a single API Management instance across any number of Azure regions. For more information, see How to deploy an Azure API Management service instance to multiple Azure regions.
- Use Traffic Manager to route HTTP requests to the primary region. If the Function App running in that region becomes unavailable, Traffic Manager can fail over to a secondary region.
- Cosmos DB supports multiple master regions, which enables writes to any region that you add to your Cosmos DB account. If you don't enable multi-master, you can still fail over the primary write region. The Cosmos DB client SDKs and the Azure Function bindings automatically handle the failover, so you don't need to update any application configuration settings.

Security considerations

Authentication

The `GetStatus` API in the reference implementation uses Azure AD to authenticate requests. Azure AD supports the OpenID Connect protocol, which is an authentication protocol built on top of the OAuth 2 protocol.

In this architecture, the client application is a single-page application (SPA) that runs in the browser. This type of client application cannot keep a client secret or an authorization code hidden, so the implicit grant flow is appropriate. (See Which OAuth 2.0 flow should I use?). Here's the overall flow:

1. The user clicks the "Sign in" link in the web application.
2. The browser is redirected the Azure AD sign in page.
3. The user signs in.
4. Azure AD redirects back to the client application, including an access token in the URL fragment.
5. When the web application calls the API, it includes the access token in the Authentication header. The application ID is sent as the audience ('aud') claim in the access token.
6. The backend API validates the access token.

To configure authentication:

- Register an application in your Azure AD tenant. This generates an application ID, which the client includes with the login URL.
- Enable Azure AD authentication inside the Function App. For more information, see Authentication and authorization in Azure App Service.
- Add the validate-jwt policy to API Management to pre-authorize the request by validating the access token.

For more details, see the GitHub readme.

It's recommended to create separate app registrations in Azure AD for the client application and the backend API. Grant the client application permission to call the API. This approach gives you the flexibility to define multiple APIs and clients and control the permissions for each.

Within an API, use scopes to give applications fine-grained control over what permissions they request from a user. For example, an API might have `Read` and `Write` scopes, and a particular client app might ask the user to authorize `Read` permissions only.

Authorization

In many applications, the backend API must check whether a user has permission to perform a given action. It's recommended to use claims-based authorization, where information about the user is conveyed by the identity provider (in this case, Azure AD) and used to make authorization decisions. For example, when you register an application in Azure AD, you can define a set of application roles. When a user signs into the application, Azure AD includes a `roles` claim for each role that the user has been granted, including roles that are inherited through group membership.

The ID token that Azure AD returns to the client contains some of the user's claims. Within the function app, these claims are available in the X-MS-CLIENT-PRINCIPAL header of the request. However, it's simpler to read this information from binding data. For other claims, use Microsoft Graph to query Azure AD. (The user must consent to this action when signing in.)

For more information, see Working with client identities.

CORS

In this reference architecture, the web application and the API do not share the same origin. That means when the application calls the API, it is a cross-origin request. Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy* and prevents a malicious site from reading sensitive

data from another site. To enable a cross-origin request, add a Cross-Origin Resource Sharing (CORS) policy to the API Management gateway:

```
<cors allow-credentials="true">
    <allowed-origins>
        <origin>[Website URL]</origin>
    </allowed-origins>
    <allowed-methods>
        <method>GET</method>
    </allowed-methods>
    <allowed-headers>
        <header>*</header>
    </allowed-headers>
</cors>
```

In this example, the **allow-credentials** attribute is **true**. This authorizes the browser to send credentials (including cookies) with the request. Otherwise, by default the browser does not send credentials with a cross-origin request.

> **Note**
>
> Be very careful about setting **allow-credentials** to **true**, because it means a website can send the user's credentials to your API on the user's behalf, without the user being aware. You must trust the allowed origin.

Enforce HTTPS

For maximum security, require HTTPS throughout the request pipeline:

- **CDN**. Azure CDN supports HTTPS on the `*.azureedge.net` subdomain by default. To enable HTTPS in the CDN for custom domain names, see Tutorial: Configure HTTPS on an Azure CDN custom domain.
- **Static website hosting**. Enable the "Secure transfer required" option on the Storage account. When this option is enabled, the storage account only allows requests from secure HTTPS connections.
- **API Management**. Configure the APIs to use HTTPS protocol only. You can configure this in the Azure portal or through a Resource Manager template:

  ```
  {
      "apiVersion": "2018-01-01",
      "type": "apis",
      "name": "dronedeliveryapi",
      "dependsOn": [
          "[concat('Microsoft.ApiManagement/service/',
  variables('apiManagementServiceName'))]"
      ],
      "properties": {
          "displayName": "Drone Delivery API",
          "description": "Drone Delivery API",
          "path": "api",
          "protocols": [ "HTTPS" ]
      },
      ...
  }
  ```

- **Azure Functions**. Enable the "HTTPS Only" setting.

Lock down the function app

All calls to the function should go through the API gateway. You can achieve this as follows:

- Configure the function app to require a function key. The API Management gateway will include the function key when it calls the function app. This prevents clients from calling the function directly, bypassing the gateway.
- The API Management gateway has a static IP address. Restrict the Azure Function to allow only calls from that static IP address. For more information, see Azure App Service Static IP Restrictions. (This feature is available for Standard tier services only.)

Protect application secrets

Don't store application secrets, such as database credentials, in your code or configuration files. Instead, use App settings, which are stored encrypted in Azure. For more information, see Security in Azure App Service and Azure Functions.

Alternatively, you can store application secrets in Key Vault. This allows you to centralize the storage of secrets, control their distribution, and monitor how and when secrets are being accessed. For more information, see Configure an Azure web application to read a secret from Key Vault. However, note that Functions triggers and bindings load their configuration settings from app settings. There is no built-in way to configure the triggers and bindings to use Key Vault secrets.

DevOps considerations

Deployment

To deploy the function app, we recommend using package files ("Run from package"). Using this approach, you upload a zip file to a Blob Storage container and the Functions runtime mounts the zip file as a read-only file system. This is an atomic operation, which reduces the chance that a failed deployment will leave the application in an inconsistent state. It can also improve cold start times, especially for Node.js apps, because all of the files are swapped at once.

API versioning

An API is a contract between a service and clients. In this architecture, the API contract is defined at the API Management layer. API Management supports two distinct but complementary versioning concepts:

- *Versions* allow API consumers to choose an API version based on their needs, such as v1 versus v2.
- *Revisions* allow API administrators to make non-breaking changes in an API and deploy those changes, along with a change log to inform API consumers about the changes.

If you make a breaking change in an API, publish a new version in API Management. Deploy the new version side-by-side with the original version, in a separate Function App. This lets you migrate existing clients to the new API without breaking client applications. Eventually, you can deprecate the previous version. API Management supports several versioning schemes: URL path, HTTP header, or query string. For more information about API versioning in general, see Versioning a RESTful web API.

For updates that are not breaking API changes, deploy the new version to a staging slot in the same Function App. Verify the deployment succeeded and then swap the staged version with the production version. Publish a revision in API Management.

Deploy the solution

To deploy the reference implementation for this architecture, see the GitHub readme.

Next steps

To learn more about the reference implementation, read Show me the code: Serverless application with Azure Functions.