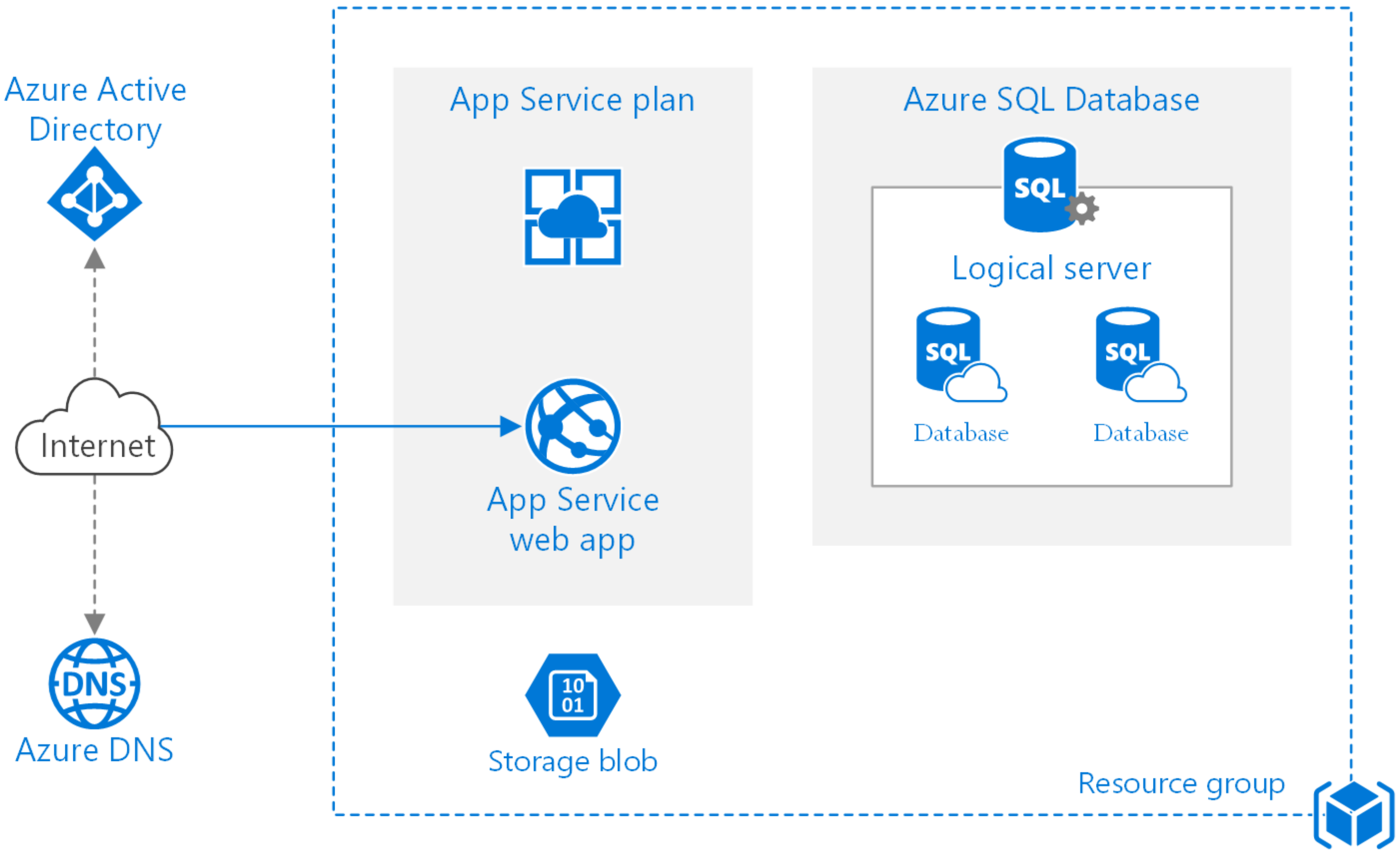
Run a basic web application in Azure

12/12/2017 • 11 minutes to read • Contributors all

In this article

This reference architecture shows proven practices for a web application that uses Azure App Service and Azure SQL Database. **Deploy this solution**.



Download a Visio file of this architecture.

Architecture

ⓘ **Note**

This architecture does not focus on application development, and does not assume any particular application framework. The goal is to understand how various Azure services fit together.

The architecture has the following components:

- **Resource group**. A resource group is a logical container for Azure resources.

- **App Service app**. Azure App Service is a fully managed platform for creating and deploying cloud applications.
- **App Service plan**. An App Service plan provides the managed virtual machines (VMs) that host your app. All apps associated with a plan run on the same VM instances.
- **Deployment slots**. A deployment slot lets you stage a deployment and then swap it with the production deployment. That way, you avoid deploying directly into production. See the Manageability section for specific recommendations.
- **IP address**. The App Service app has a public IP address and a domain name. The domain name is a subdomain of `azurewebsites.net`, such as `contoso.azurewebsites.net`.
- **Azure DNS**. Azure DNS is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services. To use a custom domain name (such as `contoso.com`) create DNS records that map the custom domain name to the IP address. For more information, see Configure a custom domain name in Azure App Service.
- **Azure SQL Database**. SQL Database is a relational database-as-a-service in the cloud. SQL Database shares its code base with the Microsoft SQL Server database engine. Depending on your application requirements, you can also use Azure Database for MySQL or Azure Database for PostgreSQL. These are fully managed database services, based on the open-source MySQL Server and Postgres database engines, respectively.
- **Logical server**. In Azure SQL Database, a logical server hosts your databases. You can create multiple databases per logical server.
- **Azure Storage**. Create an Azure storage account with a blob container to store diagnostic logs.
- **Azure Active Directory (Azure AD)**. Use Azure AD or another identity provider for authentication.

Recommendations

Your requirements might differ from the architecture described here. Use the recommendations in this section as a starting point.

App Service plan

Use the Standard or Premium tiers, because they support scale-out, autoscale, and secure sockets layer (SSL). Each tier supports several *instance sizes* that differ by number of cores and memory. You can change the tier or instance size after you create a plan. For more information about App Service plans, see App Service Pricing.

You are charged for the instances in the App Service plan, even if the app is stopped. Make sure to delete plans that you aren't using (for example, test deployments).

SQL Database

Use the V12 version of SQL Database. SQL Database supports Basic, Standard, and Premium service tiers, with multiple performance levels within each tier measured in Database Transaction Units (DTUs). Perform capacity planning and choose a tier and performance level that meets your requirements.

Region

Provision the App Service plan and the SQL Database in the same region to minimize network latency. Generally, choose the region closest to your users.

The resource group also has a region, which specifies where deployment metadata is stored. Put the resource group and its resources in the same region. This can improve availability during deployment.

Scalability considerations

A major benefit of Azure App Service is the ability to scale your application based on load. Here are some considerations to keep in mind when planning to scale your application.

Scaling the App Service app

There are two ways to scale an App Service app:

- *Scale up*, which means changing the instance size. The instance size determines the memory, number of cores, and storage on each VM instance. You can scale up manually by changing the instance size or the plan tier.

- *Scale out*, which means adding instances to handle increased load. Each pricing tier has a maximum number of instances.

  You can scale out manually by changing the instance count, or use autoscaling to have Azure automatically add or remove instances based on a schedule and/or performance metrics. Each scale operation happens quickly—typically within seconds.

  To enable autoscaling, create an autoscale *profile* that defines the minimum and maximum number of instances. Profiles can be scheduled. For example, you might create separate profiles for weekdays and weekends. Optionally, a profile contains rules for when to add or remove instances. (Example: Add two instances if CPU usage is above 70% for 5 minutes.)

Recommendations for scaling a web app:

- As much as possible, avoid scaling up and down, because it may trigger an application restart. Instead, select a tier and size that meet your performance requirements under typical load and then scale out the instances to handle changes in traffic volume.
- Enable autoscaling. If your application has a predictable, regular workload, create profiles to schedule the instance counts ahead of time. If the workload is not predictable, use rule-based autoscaling to react to changes in load as they occur. You can combine both approaches.
- CPU usage is generally a good metric for autoscale rules. However, you should load test your application, identify potential bottlenecks, and base your autoscale rules on that data.
- Autoscale rules include a *cool-down* period, which is the interval to wait after a scale action has completed before starting a new scale action. The cool-down period lets the system stabilize before scaling again. Set a shorter cool-down period for adding instances, and a longer cool-down period for removing instances. For example, set 5 minutes to add an instance, but 60 minutes to remove an instance. It's better to add new instances quickly under heavy load to handle the additional traffic, and then gradually scale back.

Scaling SQL Database

If you need a higher service tier or performance level for SQL Database, you can scale up individual databases with no application downtime. For more information, see Scale single database resources in Azure SQL Database.

Availability considerations

At the time of writing, the service level agreement (SLA) for App Service is 99.95% and the SLA for SQL Database is 99.99% for Basic, Standard, and Premium tiers.

ⓘ **Note**

The App Service SLA applies to both single and multiple instances.

Backups

In the event of data loss, SQL Database provides point-in-time restore and geo-restore. These features are available in all tiers and are automatically enabled. You don't need to schedule or manage the backups.

- Use point-in-time restore to recover from human error by returning the database to an earlier point in time.
- Use geo-restore to recover from a service outage by restoring a database from a geo-redundant backup.

For more information, see Cloud business continuity and database disaster recovery with SQL Database.

App Service provides a backup and restore feature for your application files. However, be aware that the backed-up files include app settings in plain text and these may include secrets, such as connection strings. Avoid using the App Service backup feature to back up your SQL databases because it exports the database to a SQL BACPAC file, consuming DTUs. Instead, use SQL Database point-in-time restore described above.

Manageability considerations

Create separate resource groups for production, development, and test environments. This makes it easier to manage deployments, delete test deployments, and assign access rights.

When assigning resources to resource groups, consider the following:

- Lifecycle. In general, put resources with the same lifecycle into the same resource group.
- Access. You can use role-based access control (RBAC) to apply access policies to the resources in a group.
- Billing. You can view the rolled-up costs for the resource group.

For more information, see Azure Resource Manager overview.

Deployment

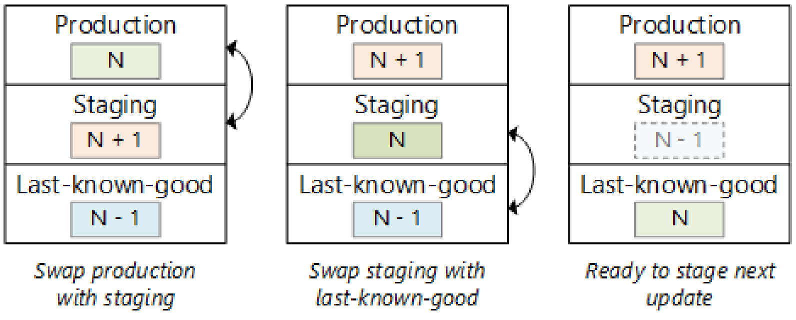
Deployment involves two steps:

1. Provisioning the Azure resources. We recommend that you use Azure Resource Manager templates for this step. Templates make it easier to automate deployments via PowerShell or the Azure CLI.
2. Deploying the application (code, binaries, and content files). You have several options, including deploying from a local Git repository, using Visual Studio, or continuous deployment from cloud-based source control. See Deploy your app to Azure App Service.

An App Service app always has one deployment slot named `production`, which represents the live production site. We recommend creating a staging slot for deploying updates. The benefits of using a staging slot include:

- You can verify the deployment succeeded, before swapping it into production.
- Deploying to a staging slot ensures that all instances are warmed up before being swapped into production. Many applications have a significant warmup and cold-start time.

We also recommend creating a third slot to hold the last-known-good deployment. After you swap staging and production, move the previous production deployment (which is now in staging) into the last-known-good slot. That way, if you discover a problem later, you can quickly revert to the last-known-good version.



If you revert to a previous version, make sure any database schema changes are backward compatible.

Don't use slots on your production deployment for testing because all apps within the same App Service plan share the same VM instances. For example, load tests might degrade the live production site. Instead, create separate App Service plans for production and test. By putting test deployments into a separate plan, you isolate them from the production version.

Configuration

Store configuration settings as app settings. Define the app settings in your Resource Manager templates, or using PowerShell. At runtime, app settings are available to the application as environment variables.

Never check passwords, access keys, or connection strings into source control. Instead, pass these as parameters to a deployment script that stores these values as app settings.

When you swap a deployment slot, the app settings are swapped by default. If you need different settings for production and staging, you can create app settings that stick to a slot and don't get swapped.

Diagnostics and monitoring

Enable diagnostics logging, including application logging and web server logging. Configure logging to use Blob storage. For performance reasons, create a separate storage account for diagnostic logs. Don't use the same storage account for logs and application data. For more detailed guidance on logging, see Monitoring and diagnostics guidance.

Use a service such as New Relic or Application Insights to monitor application performance and behavior under load. Be aware of the data rate limits for Application Insights.

Perform load testing, using a tool such as Azure DevOps or Visual Studio Team Foundation Server. For a general overview of performance analysis in cloud applications, see Performance Analysis Primer.

Tips for troubleshooting your application:

- Use the troubleshoot blade in the Azure portal to find solutions to common problems.
- Enable log streaming to see logging information in near-real time.
- The Kudu dashboard has several tools for monitoring and debugging your application. For more information, see Azure Websites online tools you should know about (blog post). You can reach the Kudu dashboard from the Azure portal. Open the blade for your app and click **Tools**, then click **Kudu**.
- If you use Visual Studio, see the article Troubleshoot a web app in Azure App Service using Visual Studio for debugging and troubleshooting tips.

Security considerations

This section lists security considerations that are specific to the Azure services described in this article. It's not a complete list of security best practices. For some additional security considerations, see Secure an app in Azure App Service.

SQL Database auditing

Auditing can help you maintain regulatory compliance and get insight into discrepancies and irregularities that could indicate business concerns or suspected security violations. See Get started with SQL database auditing.

Deployment slots

Each deployment slot has a public IP address. Secure the nonproduction slots using Azure Active Directory login so that only members of your development and DevOps teams can reach those endpoints.

Logging

Logs should never record users' passwords or other information that might be used to commit identity fraud. Scrub those details from the data before storing it.

SSL

An App Service app includes an SSL endpoint on a subdomain of `azurewebsites.net` at no additional cost. The SSL endpoint includes a wildcard certificate for the `*.azurewebsites.net` domain. If you use a custom domain name, you must provide a certificate that matches the custom domain. The simplest approach is to buy a certificate directly through the Azure portal. You can also import certificates from other certificate authorities. For more information, see Buy and Configure an SSL Certificate for your Azure App Service.

As a security best practice, your app should enforce HTTPS by redirecting HTTP requests. You can implement this inside your application or use a URL rewrite rule as described in Enable HTTPS for an app in Azure App Service.

Authentication

We recommend authenticating through an identity provider (IDP), such as Azure AD, Facebook, Google, or Twitter. Use OAuth 2 or OpenID Connect (OIDC) for the authentication flow. Azure AD provides functionality to manage users and groups, create application roles, integrate your on-premises identities, and consume backend services such as Office 365 and Skype for Business.

Avoid having the application manage user logins and credentials directly, as it creates a potential attack surface. At a minimum, you would need to have email confirmation, password recovery, and multi-factor authentication; validate password strength; and store password hashes securely. The large identity providers handle all of those things for you, and are constantly monitoring and improving their security practices.

Consider using App Service authentication to implement the OAuth/OIDC authentication flow. The benefits of App Service authentication include:

- Easy to configure.
- No code is required for simple authentication scenarios.
- Supports delegated authorization using OAuth access tokens to consume resources on behalf of the user.
- Provides a built-in token cache.

Some limitations of App Service authentication:

- Limited customization options.
- Delegated authorization is restricted to one backend resource per login session.
- If you use more than one IDP, there is no built-in mechanism for home realm discovery.
- For multi-tenant scenarios, the application must implement the logic to validate the token issuer.

Deploy the solution

An example Resource Manager template for this architecture is available on GitHub.

To deploy the template using PowerShell, run the following commands:

PowerShell

```
New-AzureRmResourceGroup -Name <resource-group-name> -Location "West US"

$parameters = @{"appName"="<app-
name>";"environment"="dev";"locationShort"="uw";"databaseName"="app-db";"administratorLogin"="
<admin>";"administratorLoginPassword"="<password>"}

New-AzureRmResourceGroupDeployment -Name <deployment-name> -ResourceGroupName <resource-group-
name> -TemplateFile .\PaaS-Basic.json -TemplateParameterObject  $parameters
```

For more information, see Deploy resources with Azure Resource Manager templates.