# Azure App Service

Azure App Service is an **HTTP-based** service for **hosting web applications, REST APIs, and mobile back ends**. You can develop in your favorite programming language, be **it .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python**. Applications **run** and **scale** with ease on **both Windows and Linux**-based environments.

## Features

### Built-in auto scale support

Baked into Azure App Service is the ability to **scale up/down or scale out/in.** Depending on the usage of the web app, you can scale the resources of the underlying machine that is hosting your web app **up/down . Resources include the number of cores or the amount of RAM available**.

**Scaling out/in** is the ability to increase, or decrease, **the number of machine instances that are running your web app.**

### Continuous integration/deployment support

The Azure portal provides out-of-the-box continuous integration and deployment **with Azure DevOps, GitHub, Bitbucket, FTP, or a local Git repo**sitory on your development machine.

### Deployment slots

You can use a **separate deployment slot other than the default production slot** when you're running in the **Standard, Premium, or Isolated App Service plan tier**.

Deployment slots are **live apps** with their own host names to be **used as staging slots**.

App content and configurations elements **can be** **swapped between two deployment slots**, including the production slot.

### App Service on Linux

App Service can also **host web apps natively on Linux for supported application stacks**. It can also **run custom Linux containers** (also known as Web App for Containers).

App Service on Linux **supports a number of language specific built-in images**. Just deploy your code.

Supported languages **include: Node.js, Java (JRE 8 & JRE 11), PHP, Python, .NET Core, and Ruby**.

If the runtime of your application requires is not supported in the built-in images, you can deploy it with a **custom container**.

#### Limitations

1. App Service on **Linux is not supported on Shared pricing tier.**
2. You **can NOT mix Windows and Linux apps** in the same **App Service plan**.
3. **Historically,** **you could not mix Windows and Linux apps** in the same **resource group**. However, all resource groups created on or after **January 21, 2021** do support this scenario. Support for resource groups created before January 21, 2021 will be rolled out across Azure regions (including National cloud regions) soon.
4. **The Azure portal shows only features that currently work for Linux apps.** As features are enabled, they're activated on the portal.

## Azure App Service plans

An App Service plan defines a set of compute resources for a web app to run.

One or more apps can be configured to run on the same computing resources (or in the same App Service plan).

In addition, Azure Functions also has the option of running in an App Service plan.

### Each App Service plan defines:

* Region (West US, East US, etc.)
* Number of VM instances
* Size of VM instances (Small, Medium, Large)
* Pricing tier (Free, Shared, Basic, Standard, Premium, PremiumV2, PremiumV3, Isolated, consumption)

### Categories of pricing tiers:

* **Shared compute**: Both **Free** and **Shared** share the resource pools of your apps with the apps of other customers. These tiers allocate CPU quotas to each app that runs on the shared resources, and the resources can't scale out.
* **Dedicated compute**: The **Basic**, **Standard**, **Premium**, **PremiumV2**, and **PremiumV3** tiers run apps on dedicated Azure VMs. Only apps in the same App Service plan share the same compute resources. The higher the tier, the more VM instances are available to you for scale-out.
* **Isolated**: This tier runs dedicated Azure VMs on dedicated Azure Virtual Networks. It provides network isolation on top of compute isolation to your apps. It provides the maximum scale-out capabilities.
* **Consumption:** This tier is only available to *function apps*. It scales the functions dynamically depending on workload.

### How does my app run and scale?

In the **Free** and **Shared** tiers, an app receives CPU minutes on a shared VM instance and can't scale out.

In other tiers, an app runs and scales as follows:

* An app runs on all the VM instances configured in the App Service plan.
* If multiple apps are in the same App Service plan, they all share the same VM instances.
* If you have multiple deployment slots for an app, all deployment slots also run on the same VM instances.
* If you enable diagnostic logs, perform backups, or run WebJobs, they also use CPU cycles and memory on these VM instances.

App Service plan is the scale unit of the App Service apps. If the plan is configured to run five VM instances, then all apps in the plan run on all five instances.

If the plan is configured for autoscaling, then all apps in the plan are scaled out together based on the autoscale settings.

### What if my app needs more capabilities or features?

**Your App Service plan can be scaled up and down at any time by changing the pricing tier of the plan.**

### Isolate your app into a new App Service plan when:

The app is resource-intensive.

You want to scale the app independently from the other apps in the existing plan.

The app needs resource in a different geographical region.

## Deploy to App Service

App Service supports both automated and manual deployment.

### Automated deployment

Automated deployment, or **continuous integration**, is a process used to push out new features and bug fixes in a fast and repetitive pattern with minimal impact on end users.

Azure supports automated deployment directly from several sources.

* **Azure DevOps**: You can push your code to Azure DevOps, build your code in the cloud, run the tests, generate a release from the code, and finally, push your code to an Azure Web App.
* **GitHub**: Azure supports automated deployment directly from GitHub. When you connect your GitHub repository to Azure for automated deployment, any changes you push to your production branch on GitHub will be automatically deployed for you.
* **Bitbucket**: With its similarities to GitHub, you can configure an automated deployment with Bitbucket.

### Manual deployment

There are a few options that you can use to **manually push your code to Azure**:

* **Git**: App Service web apps feature a Git URL that you can add as a remote repository. Pushing to the remote repository will deploy your app.
* **CLI**: webapp up is a feature of the **az**command-line interface that packages your app and deploys it. Unlike other deployment methods**, az webapp up** can create a new App Service web app for you if you haven't already created one.
* **Zip deploy**: Use curl or a similar HTTP utility to send a ZIP of your application files to App Service.
* **FTP/S**: FTP or FTPS is a traditional way of pushing your code to many hosting environments, including App Service.

### Use deployment slots

Whenever possible, use deployment slots when deploying a new production build.

The swap operation warms up the necessary worker instances to match your production scale, thus eliminating downtime.

## Explore authentication and authorization in App Service

Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your web app, API, and mobile back end, and also Azure Functions.

### Why use the built-in authentication?

**You're not required to use App Service for authentication and authorization**.

Many web frameworks are bundled with security features, and you can use them if you like.

If you need more flexibility than App Service provides, you can also write your own utilities.

The built-in authentication feature for App Service and Azure Functions can save you time and effort by providing out-of-the-box authentication with federated identity providers, allowing you to focus on the rest of your application.

Azure App Service allows you to integrate a variety of auth capabilities into your web app or API without implementing them yourself.

It’s built directly into the platform and doesn’t require any particular language, SDK, security expertise, or even any code to utilize.

You can integrate with multiple login providers. For example, Azure AD, Facebook, Google, Twitter, Any OpenID Connect provider.

### Identity providers

| **Provider** | **Sign-in endpoint** | **How-To guidance** |
| --- | --- | --- |
| Microsoft Identity Platform | /.auth/login/aad | [App Service Microsoft Identity Platform login](https://docs.microsoft.com/en-us/azure/app-service/configure-authentication-provider-aad) |
| Facebook | /.auth/login/facebook | [App Service Facebook login](https://docs.microsoft.com/en-us/azure/app-service/configure-authentication-provider-facebook) |
| Google | /.auth/login/google | [App Service Google login](https://docs.microsoft.com/en-us/azure/app-service/configure-authentication-provider-google) |
| Twitter | /.auth/login/twitter | [App Service Twitter login](https://docs.microsoft.com/en-us/azure/app-service/configure-authentication-provider-twitter) |
| Any OpenID Connect provider | /.auth/login/<providerName> | [App Service OpenID Connect login](https://docs.microsoft.com/en-us/azure/app-service/configure-authentication-provider-openid-connect) |

### How it works

This module handles several things for your app:

* Authenticates users with the specified provider
* Validates, stores, and refreshes tokens
* Manages the authenticated session
* Injects identity information into request headers

### Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK.

* **Without provider SDK:** The application delegates federated sign-in to App Service. This is typically the case with browser apps, which can present the provider's login page to the user. The server code manages the sign-in process, so it is also called *server-directed flow* or *server flow*.
* **With provider SDK:** The application signs users into the provider manually and then submits the authentication token to App Service for validation. This is typically the case with browser-less apps, which can't present the provider's sign-in page to the user. The application code manages the sign-in process, so it is also called *client-directed flow* or *client flow*. This applies to REST APIs, Azure Functions, JavaScript browser clients, and native mobile apps that sign users in using the provider's SDK.

| **Step** | **Without provider SDK** | **With provider SDK** |
| --- | --- | --- |
| Sign user in | Redirects client to /.auth/login/<provider>. | Client code signs user in directly with provider's SDK and receives an authentication token. For information, see the provider's documentation. |
| Post-authentication | Provider redirects client to /.auth/login/<provider>/callback. | Client code posts token from provider to /.auth/login/<provider> for validation. |
| Establish authenticated session | App Service adds authenticated cookie to response. | App Service returns its own authentication token to client code. |
| Serve authenticated content | Client includes authentication cookie in subsequent requests (automatically handled by browser). | Client code presents authentication token in X-ZUMO-AUTH header (automatically handled by Mobile Apps client SDKs). |

You can also present users with one or more “/.auth/login/<provider>” links to sign in to your app using their provider of choice.

### Authentication behavior

You can configure App Service with a few behaviors when an incoming request is not authenticated.

* **Allow unauthenticated requests:** This option defers authorization of unauthenticated traffic to your application code. For authenticated requests, App Service also passes along authentication information in the HTTP headers.
* **Require authentication:** This option will reject any unauthenticated traffic to your application. This rejection can be a redirect action to one of the configured identity providers. In these cases, a browser client is redirected to /.auth/login/<provider> for the provider you choose.

If the anonymous request comes from a native mobile app, the returned response is an HTTP 401 Unauthorized. You can also configure the rejection to be an HTTP 401 Unauthorized or HTTP 403 Forbidden for all requests.

## App Service networking features

By default, apps hosted in App Service are accessible directly through the internet and can reach only internet-hosted endpoints.

But for many applications, you need to control the inbound and outbound network traffic.

The multitenant public service hosts App Service plans in the Free, Shared, Basic, Standard, Premium, PremiumV2, and PremiumV3 pricing SKUs.

There is also the single-tenant App Service Environment (ASE) hosts Isolated SKU App Service plans directly in your Azure virtual network.

### Multi-tenant App Service networking features

Azure App Service is a distributed system. The roles that handle incoming HTTP or HTTPS requests are called front ends. The roles that host the customer workload are called workers. All the roles in an App Service deployment exist in a multi-tenant network.

| **Inbound features** | **Outbound features** |
| --- | --- |
| App-assigned address | Hybrid Connections |
| Access restrictions | Gateway-required virtual network integration |
| Service endpoints | Virtual network integration |
| Private endpoints |  |

# Autoscale apps in Azure App Service

## What is Autoscaling?

Autoscaling is a cloud system or process that adjusts available resources based on the current demand. Autoscaling performs scaling in and out (adding or removing web servers), as opposed to scaling up and down.

## Azure App Service Autoscaling

Autoscaling in Azure App Service monitors the resource metrics of a web app as it runs. It detects situations where additional resources are required to handle an increasing workload, and ensures those resources are available before the system becomes overloaded.

Autoscaling responds to changes in the environment by adding or removing web servers and balancing the load between them. Autoscaling doesn't have any effect on the CPU power, memory, or storage capacity of the web servers powering the app, it only changes the number of web servers.

## Autoscaling rules

Autoscaling makes its decisions based on rules that you define. A rule specifies the threshold for a metric, and triggers an Autoscale event when this threshold is crossed. Autoscaling can also deallocate resources when the workload has diminished.

Define your autoscaling rules carefully. For example, a Denial of Service attack will likely result in a large-scale influx of incoming traffic. Trying to handle a surge in requests caused by a DoS attack would be fruitless and expensive. These requests aren't genuine, and should be discarded rather than processed. 7A better solution is to implement detection and filtering of requests that occur during such an attack before they reach your service.

## When should you consider autoscaling?

Autoscaling provides elasticity for your services. It's a suitable solution when hosting any application when you can't easily predict the workload in advance, or when the workload is likely to vary by date or time.

Autoscaling improves availability and fault tolerance. It can help ensure that client requests to a service won't be denied because an instance is either not able to acknowledge the request in a timely manner, or because an overloaded instance has crashed.

Autoscaling is NOT the best approach to handling long-term growth.

The initial number of instances of a service is also a factor. The fewer the number of instances initially, the less capacity you have to handle an increasing workload while autoscaling spins up additional instances.

## Autoscale Factors

Autoscaling enables you to specify the conditions under which a web app should be scaled out, and back in again. Effective autoscaling ensures sufficient resources are available to handle large volumes of requests at peak times, while managing costs when the demand drops.

You can configure autoscaling conditions based on resource usage or according to a schedule.

### Autoscaling and the App Service Plan

Autoscaling is a feature of the App Service Plan used by the web app. When the web app scales out, Azure starts new instances of the hardware defined by the App Service Plan to the app.

To prevent runaway autoscaling, an App Service Plan has an instance limit. Plans in more expensive pricing tiers have a higher limit. Autoscaling cannot create more instances than this limit.y

### Autoscale conditions

You indicate how to autoscale by creating autoscale conditions. Azure provides two options for autoscaling:

* Scale based on a metric, such as the length of the disk queue, or the number of HTTP requests awaiting processing.
* Scale to a specific instance count according to a schedule. For example, you can arrange to scale out at a particular time of day, or on a specific date or day of the week. You also specify an end date, and the system will scale back in at this time.

You can also use combination of above two as autoscale conditions.

### Metrics for autoscale rules

An autoscale rule specifies a metric to monitor, and how autoscaling should respond when this metric crosses a defined threshold.

 The metrics you can monitor for a web app are:

* **CPU Percentage**. This metric is an indication of the CPU utilization across all instances. A high value shows that instances are becoming CPU-bound, which could cause delays in processing client requests.
* **Memory Percentage**. This metric captures the memory occupancy of the application across all instances. A high value indicates that free memory could be running low, and could cause one or more instances to fail.
* **Disk Queue Length**. This metric is a measure of the number of outstanding I/O requests across all instances. A high value means that disk contention could be occurring.
* **Http Queue Length**. This metric shows how many client requests are waiting for processing by the web app. If this number is large, client requests might fail with HTTP 408 (Timeout) errors.
* **Data In**. This metric is the number of bytes received across all instances.
* **Data Out**. This metric is the number of bytes sent by all instances.

You can also scale based on metrics for other Azure services. For example, if the web app processes requests received from a Service Bus Queue, you might want to spin up additional instances of a web app if the number of items held in an Azure Service Bus Queue exceeds a critical length.

### How an autoscale rule analyzes metrics

Each metric has its own intrinsic time grain, but in most cases this period is 1 minute.

Autoscale rules perform aggregation of the value calculated by the time grain over a longer, user-specified period, known as the Duration. The minimum Duration is 5 minutes.

### Autoscale actions

When an autoscale rule detects that a metric has crossed a threshold, it can perform an autoscale action. An autoscale action can be scale-out or scale-in.

### Pairing autoscale rules

One autoscale rule should indicate how to scale the system out when a metric exceeds an upper threshold.

Then other rule should define how to scale the system back in again when the same metric drops below a lower threshold.

### Combining autoscale rules

A single autoscale condition can contain several autoscale rules (for example, a scale-out rule and the corresponding scale-in rule). However, the autoscale rules in an autoscale condition don't have to be directly related. You could define the following four rules in the same autoscale condition:

* If the HTTP queue length exceeds 10, scale out by 1
* If the CPU utilization exceeds 70%, scale out by 1
* If the HTTP queue length is zero, scale in by 1
* If the CPU utilization drops below 50%, scale in by 1

## Enable autoscaling

Azure portal -> your App Service plan -> Left menu -> Settings -> Scale out (App Service plan).

By default, an App Service Plan only implements manual scaling. Selecting Custom autoscale reveals condition groups you can use to manage your scale settings.

### Add scale conditions

Once you enable autoscaling, you can edit the automatically created default scale condition, and you can add your own custom scale conditions. Remember that each scale condition can either scale based on a metric, or scale to a specific instance count.

The Default scale condition is executed when none of the other scale conditions are active.

### Create scale rules

You use the Add a rule link to add your own custom rules. You define the criteria that indicate when a rule should trigger an autoscale action, and the autoscale action to be performed (scale out or scale in).

### Monitor autoscaling activity

The Azure portal enables you to track when autoscaling has occurred through the Run history chart. This chart shows how the number of instances varies over time, and which autoscale conditions caused each change.

You can use the Run history chart in conjunction with the metrics shown on the Overview page to correlate the autoscaling events with resource utilization.

## Autoscale Best Practices

If you're not following good practices when creating autoscale settings, you can create conditions that lead to undesirable results. You need to avoid creating rules that conflict with each other.

### Autoscale concepts

* An autoscale setting scales instances horizontally, which is *out* by increasing the instances and *in* by decreasing the number of instances. An autoscale setting has a maximum, minimum, and default value of instances.
* An autoscale job always reads the associated metric to scale by, checking if it has crossed the configured threshold for scale out or scale-in.
* All thresholds are calculated at an instance level. For example, "scale out by one instance when average CPU > 80% when instance count is 2", means scale out when the average CPU across all instances is greater than 80%.
* All autoscale successes and failures are logged to the Activity Log. You can then configure an activity log alert so that you can be notified via email, SMS, or webhooks whenever there's activity.

### Autoscale best practices

Use the following best practices as you create your autoscale rules.

#### Ensure the maximum and minimum values are different and have an adequate margin between them

If you have a setting that has minimum=two, maximum=two and the current instance count is two, no scale action can occur. Keep an adequate margin between the maximum and minimum instance counts, which are inclusive. Autoscale always scales between these limits.

#### Choose the appropriate statistic for your diagnostics metric

For diagnostics metrics, you can choose among *Average*, *Minimum*, *Maximum* and *Total* as a metric to scale by. The most common statistic is Average.

#### Choose the thresholds carefully for all metric types

We recommend carefully choosing different thresholds for scale out and scale-in based on practical situations.

We don't recommend autoscale settings like the examples below with the same or similar threshold values for out and in conditions:

* Increase instances by one count when Thread Count >= 600
* Decrease instances by one count when Thread Count <= 600

Let's look at an example of what can lead to a behavior that may seem confusing. Consider the following sequence.

Assume there are two instances to begin with and then the average number of threads per instance grows to 625.

Autoscale scales out adding a third instance.

Next, assume that the average thread count across instance falls to 575.

Before scaling in, autoscale tries to estimate what the final state will be if it scaled in. For example, 575 x 3 (current instance count) = 1,725 / 2 (final number of instances when scaled in) = 862.5 threads. This means autoscale would have to immediately scale out again even after it scaled in, if the average thread count remains the same or even falls only a small amount. However, if it scaled out again, the whole process would repeat, leading to an infinite loop.

**To avoid** **this situation (termed "flapping")**, autoscale doesn't scale in at all. Instead, it skips and reevaluates the condition again the next time the service's job executes. This can confuse many people because autoscale wouldn't appear to work when the average thread count was 575

### Considerations for scaling when multiple rules are configured in a profile

If you have the following four autoscale rules:

* If CPU < 30 %, scale-in by 1
* If Memory < 50%, scale-in by 1
* If CPU > 75%, scale out by 1
* If Memory > 75%, scale out by 1

Then the follow occurs:

* If CPU is 76% and Memory is 50%, we scale out.
* If CPU is 50% and Memory is 76% we scale out.

### Always select a safe default instance count

The default instance count is important because autoscale scales your service to that count when metrics aren't available. Therefore, select a default instance count that's safe for your workloads.

### Configure autoscale notifications

Autoscale will post to the Activity Log if any of the following conditions occur:

* Autoscale issues a scale operation
* Autoscale service successfully completes a scale action
* Autoscale service fails to take a scale action.
* Metrics aren't available for autoscale service to make a scale decision.
* Metrics are available (recovery) again to make a scale decision.

You can also use an Activity Log alert to monitor the health of the autoscale engine. In addition to using activity log alerts, you can also configure email or webhook notifications to get notified for successful scale actions via the notifications tab on the autoscale setting.

# Azure App Service deployment slots

## Explore staging environments

When you deploy your web app, web app on Linux, mobile back end, or API app to Azure App Service, you can use a separate deployment slot instead of the default production slot when you're running in the Standard, Premium, or Isolated App Service plan tier.

Deployment slots are live apps with their own host names. App content and configurations elements can be swapped between two deployment slots, including the production slot.

Deploying your application to a non-production slot has the **following benefits:**

* **You can validate app changes in a staging** deployment slot before swapping it with the production slot.
* Deploying an app to a slot first and swapping it into production makes sure that **all instances of the slot are warmed up before** being swapped into production. This eliminates downtime when you deploy your app. **The traffic redirection is seamless**, and no requests are dropped because of swap operations.
  + *You can* ***automate*** *this entire workflow by configuring auto swap when pre-swap validation isn't needed*.
* After a swap, the slot with previously staged app now has the previous production app. If the changes swapped into the production slot aren't as you expect, you can perform the same swap **immediately to get your "last known good site" back**.

Each App Service plan tier supports a different number of deployment slots. There's no additional charge for using deployment slots. To find out the number of slots your app's tier supports, visit [App Service limits](https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/azure-subscription-service-limits#app-service-limits).

When you create a new slot the new deployment slot has no content, even if you clone the settings from a different slot. You can deploy to the slot from a different repository branch or a different repository.

## Examine slot swapping

When you swap slots (for example, from a staging slot to the production slot), **App Service does the following** to ensure that the target slot doesn't experience downtime:

1. Apply the following settings from the target slot (for example, the production slot) to all instances of the source slot:
   * Slot-specific app settings and connection strings, if applicable.
   * Continuous deployment settings, if enabled.
   * App Service authentication settings, if enabled.

Any of these cases trigger all instances in the source slot to restart. During **swap with preview**, this marks the end of the first phase. The swap operation is paused, and you can validate that the source slot works correctly with the target slot's settings.

1. Wait for every instance in the source slot to complete its restart. If any instance fails to restart, the swap operation reverts all changes to the source slot and stops the operation.
2. If local cache is enabled, trigger local cache initialization by making an HTTP request to the application root ("/") on each instance of the source slot. Wait until each instance returns any HTTP response. Local cache initialization causes another restart on each instance.
3. If auto swap is enabled with custom warm-up, trigger Application Initiation by making an HTTP request to the application root ("/") on each instance of the source slot.
   * If applicationInitialization isn't specified, trigger an HTTP request to the application root of the source slot on each instance.
   * If an instance returns any HTTP response, it's considered to be warmed up.
4. If all instances on the source slot are warmed up successfully, swap the two slots by switching the routing rules for the two slots. After this step, the target slot (for example, the production slot) has the app that's previously warmed up in the source slot.
5. Now that the source slot has the pre-swap app previously in the target slot, perform the same operation by applying all settings and restarting the instances.

At any point of the swap operation, **all work** of initializing the swapped apps **happens on the source slot**. The **target slot remains online** while the **source slot is being prepared and warmed up**, regardless of where the swap succeeds or fails.

To swap a staging slot with the production slot, **make sure that the production slot is always the target slot**. This way, the swap operation doesn't affect your production app.

| **Settings that are swapped** | **Settings that aren't swapped** |
| --- | --- |
| General settings, such as framework version, 32/64-bit, web sockets | Publishing endpoints |
| App settings (can be configured to stick to a slot) | Custom domain names |
| Connection strings (can be configured to stick to a slot) | Non-public certificates and TLS/SSL settings |
| Handler mappings | Scale settings |
| Public certificates | WebJobs schedulers |
| WebJobs content | IP restrictions |
| Hybrid connections \* | Always On |
| Virtual network integration \* | Diagnostic log settings |
| Service endpoints \* | Cross-origin resource sharing (CORS) |
| Azure Content Delivery Network \* |  |

Features marked with an asterisk (\*) are planned to be unswapped.

***To make settings swappable, add the app setting WEBSITE\_OVERRIDE\_PRESERVE\_DEFAULT\_STICKY\_SLOT\_SETTINGS in every slot of the app and set its value to 0 or false. These settings are either all swappable or not at all. You can't make just some settings swappable and not the others. Managed identities are never swapped and are not affected by this override app setting.***

To configure an app setting or connection string to stick to a specific slot (not swapped), go to the Configuration page for that slot. Add or edit a setting, and then select Deployment slot setting. Selecting this check box tells App Service that the setting is not swappable.

## Swap deployment slots

You can swap deployment slots on your app's Deployment slots page and the Overview page.

Before you swap an app from a deployment slot into production, make sure that production is your target slot and that all settings in the source slot are configured exactly as you want to have them in production.

### Manually swapping deployment slots

To swap deployment slots:

1. Go to your app's **Deployment slots** page and select **Swap**. The **Swap** dialog box shows settings in the selected source and target slots that will be changed.
2. Select the desired **Source** and **Target** slots. Usually, the target is the production slot. Also, select the **Source Changes** and **Target Changes** tabs and verify that the configuration changes are expected. When you're finished, you can swap the slots immediately by selecting **Swap**.

To see how your target slot would run with the new settings before the swap actually happens, don't select Swap, but follow the instructions in *Swap with preview* below.

1. When you're finished, Close.

### Swap with preview (multi-phase swap)

When you perform a swap with preview, App Service performs the same swap operation but pauses after the first step. You can then verify the result on the staging slot before completing the swap.

If you cancel the swap, App Service reapplies configuration elements to the source slot.

**To swap with preview:**

1. Follow the steps above in Swap deployment slots but select **Perform swap with preview**. The dialog box shows you how the configuration in the source slot changes in phase 1, and how the source and target slot change in phase 2.
2. When you're ready to start the swap, select **Start Swap**.

When phase 1 finishes, you're notified in the dialog box. Preview the swap in the source slot by going to https://<app\_name>-<source-slot-name>.azurewebsites.net.

1. When you're ready to complete the pending swap, select **Complete Swap** in **Swap action** and select **Complete Swap**.

To cancel a pending swap, select **Cancel Swap** instead.

1. When you're finished, close the dialog box by selecting **Close**.

### Configure auto swap

Auto swap streamlines Azure DevOps scenarios where you want to deploy your app continuously with zero cold starts and zero downtime for customers of the app. When auto swap is enabled from a slot into production, every time you push your code changes to that slot, App Service automatically swaps the app into production after it's warmed up in the source slot.

(Auto swap isn't currently supported in web apps on Linux).

### To configure auto swap:

1. Go to your app's resource page and select the deployment slot you want to configure to auto swap. The setting is on the **Configuration > General settings** page.
2. Set **Auto swap enabled** to **On**. Then select the desired target slot for Auto swap deployment slot, and select **Save** on the command bar.
3. Execute a code push to the source slot. Auto swap happens after a short time, and the update is reflected at your target slot's URL.

### Specify custom warm-up

Some apps might require custom warm-up actions before the swap. The applicationInitialization configuration element in web.config lets you specify custom initialization actions. The swap operation waits for this custom warm-up to finish before swapping with the target slot.

Here's a sample web.config fragment.

<system.webServer>

<applicationInitialization>

<add initializationPage="/" hostName="[app hostname]" />

<add initializationPage="/Home/About" hostName="[app hostname]" />

</applicationInitialization>

</system.webServer>

You can also customize the warm-up behavior with one or both of the following app settings:

* **WEBSITE\_SWAP\_WARMUP\_PING\_PATH**: The path to ping to warm up your site. Add this app setting by specifying a custom path that begins with a slash as the value. An example is /statuscheck. The default value is /.
* **WEBSITE\_SWAP\_WARMUP\_PING\_STATUSES**: Valid HTTP response codes for the warm-up operation. Add this app setting with a comma-separated list of HTTP codes. An example is 200,202 . If the returned status code isn't in the list, the warmup and swap operations are stopped. By default, all response codes are valid.

### Roll back and monitor a swap

If any errors occur in the target slot (for example, the production slot) after a slot swap, restore the slots to their pre-swap states by swapping the same two slots immediately.

If the swap operation takes a long time to complete, you can get information on the swap operation in the activity log.

On your app's resource page in the portal, in the left pane, select Activity log.

A swap operation appears in the log query as Swap Web App Slots. You can expand it and select one of the suboperations or errors to see the details.

## Route traffic in App Service

By default, all client requests to the app's production URL (http://<app\_name>.azurewebsites.net) are routed to the production slot. You can route a portion of the traffic to another slot. This feature is useful if you need user feedback for a new update, but you're not ready to release it to production.

### Route production traffic automatically

To route production traffic automatically:

1. Go to your app's resource page and select **Deployment slots**.
2. In the **Traffic %** column of the slot you want to route to, specify a percentage (between 0 and 100) to represent the amount of total traffic you want to route. Select **Save**.

After the setting is saved, the specified percentage of clients is randomly routed to the non-production slot.

### Route production traffic manually

In addition to automatic traffic routing, App Service can route requests to a specific slot. This is useful when you want your users to be able to opt in to or opt out of your beta app. To route production traffic manually, you use the ***x-ms-routing-name*** query parameter.

**By default, new slots are given a routing rule of 0%, a default value is displayed in grey. When you explicitly set this value to 0% it is displayed in black, your users can access the staging slot manually by using the *x-ms-routing-name* query parameter. But they won't be routed to the slot automatically because the routing percentage is set to 0. This is an advanced scenario where you can "hide" your staging slot from the public while allowing internal teams to test changes on the slot.**