# Overview

GitHub hosts over 25 million repositories containing applications of all shapes and sizes. But GitHub is just a start—those applications still need to get built, released, and managed to reach their full potential.

With the introduction of [Azure DevOps](https://azure.com/devops), Microsoft offers developers a new continuous integration/continuous delivery (CI/CD) service called [Azure Pipelines](https://azure.microsoft.com/services/devops/pipelines/) that enables you to continuously build, test, and deploy to any platform or cloud. It has cloud-hosted agents for Linux, macOS, and Windows; powerful workflows with native container support; and flexible deployments to Kubernetes, VMs, and serverless environments.

Azure Pipelines provides unlimited CI/CD minutes and 10 parallel jobs to every GitHub open source project for free. All open source projects run on the same infrastructure that our paying customers use. That means you’ll have the same fast performance and high quality of service. Many of the top open source projects are already using Azure Pipelines for CI/CD, such as Atom, CPython, Pipenv, Tox, Visual Studio Code, and TypeScript—and the list is growing every day.

In addition to Azure Pipelines, GitHub users can also benefit from [Azure Boards](https://azure.microsoft.com/services/devops/boards/), a set of features that enable you to plan, track, and discuss work across your teams using Kanban boards, backlogs, team dashboards, and custom reporting. You can link GitHub activities from Azure Boards by mentioning them in commits and pull requests, and even automate the state transition of linked work items when pull requests are approved.

In this demo, you’ll see how easy it is to set up Azure Pipelines and Azure Boards with your GitHub projects and how you can start seeing benefits immediately.

# Key Takeaways

The key takeaways of the demo are:

* Microsoft provides the only comprehensive DevOps solution that spans from development to project management to deployment to operations.
* It doesn’t matter what technologies of processes you’re using—even setting up a Node.js solution on GitHub to deploy to a Linux container that connects to a Cosmos DB is a seamless, straightforward experience.
* Azure offers a practical approach to automation at every step of the DevOps lifecycle that enables companies to focus their efforts on creating business value.

# ## Prerequisites

These items are required for this demo.

1. A GitHub account from <https://github.com>.
2. An Azure account from <https://azure.com>.
3. An Azure DevOps account from <https://dev.azure.com>.
4. ARM Outputs extension installed in your Azure DevOps account from <https://marketplace.visualstudio.com/items?itemName=keesschollaart.arm-outputs>.
5. Git installed from <https://git-scm.com/downloads>.
6. Visual Studio Code installed from <https://code.visualstudio.com>.
7. Azure Pipelines extension for Visual Studio Code installed from <https://marketplace.visualstudio.com/items?itemName=ms-azure-devops.azure-pipelines>.
8. GitHub Pull Requests extension for Visual Studio Code installed from <https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github>.

## Demo Setup

You will need to perform these steps prior to presenting this demo.

1. Clone the GitHub repo locally and open it in Visual Studio Code.
2. Create a new Azure DevOps project, preferably named something like “ContosoAir”.
3. Have two separate browser tabs open and logged in: one on the GitHub project root and one on the Azure portal.

============

## Demo Scenario

In this demo, we’ll be illustrating the integration and automation benefits of Azure DevOps. We will take on the role of helping a fictitious airline—Contoso Air—that has developed their flagship web site using Node.js. To improve their operations, they want to implement pipelines for continuous integration and continuous delivery so that they can quickly update their public services and take advantage of the full benefits of DevOps and the cloud.

The site will be hosted in Azure, and they want to automate the entire process so that they can spin up all the infrastructure needed to deploy and host the application without any manual intervention. Once this process is in place, it will free up their technology teams to focus more on generating business value.

## Task 1 – Installing Azure Pipelines

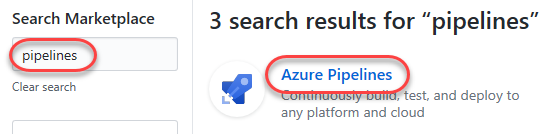
1. Fork the GitHub project at [https://github.com/Microsoft/ContosoAir/](https://na01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fgithub.com%2FMicrosoft%2FContosoAir%2F&data=02%7C01%7Csraj%40microsoft.com%7Cad53721c5cd84d53603808d6436a6fe4%7C72f988bf86f141af91ab2d7cd011db47%7C1%7C0%7C636770521758495618&sdata=clPq649M5gB1RKos1J6sZZcQrPFUV7CSwUvle6EYlp8%3D&reserved=0).
2. Switch to the browser tab open to the root of your GitHub fork. It should be something like <https://github.com/account/ContosoAir>.

* > **Talk track:** In this demo, we will help Contoso Air revamp a critical component of their DevOps scenario. Like all airlines, they rely on their web site to generate and manage business opportunities. However, the current processes they have in place to move a change from their source code to their production systems is time-consuming and open to human error. They use GitHub to manage their source code and want to host their production site on Azure, so it will be our job to automate everything in the middle. This will involve setting up a pipeline so that commits to the GitHub repo invoke a continuous integration build in Azure DevOps. Once that build is complete, it will invoke a continuous delivery deployment to push the bits out to Azure, creating the required resources, if necessary. The first thing we need to do is to connect GitHub with Azure DevOps, which we can do via the Azure Pipelines extension in the GitHub Marketplace.

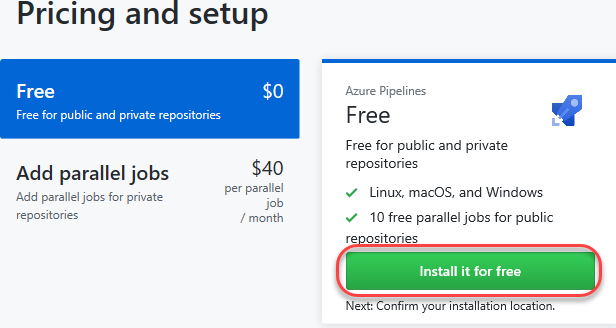
1. Navigate to the **GitHub Marketplace**.

* 

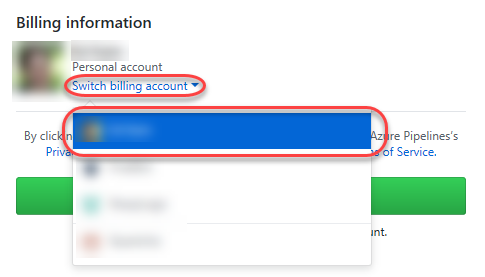
1. Search for **“pipelines”** and click **Azure Pipelines**.

* 
* > **Talk track:** Azure Pipelines is free to use for both public and private repos. If you have a need to scale your builds, you can add parallel job support for a nominal fee. Installing it into your GitHub account involves just a few clicks, and you can configure exactly which repos you want to grant it access to.

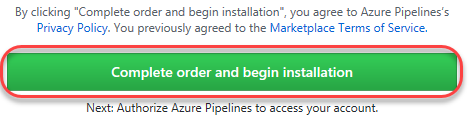
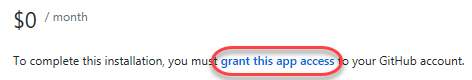
1. Scroll to the bottom and click **Install it for free**.

* 

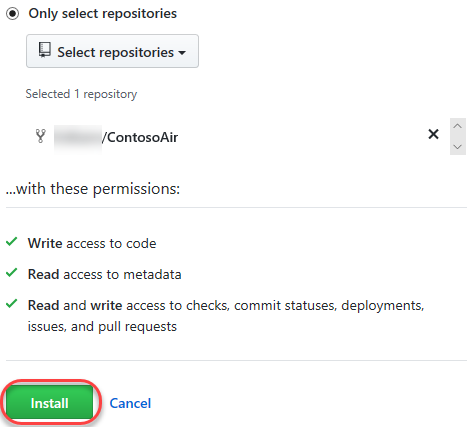
1. If you have multiple **GitHub** accounts, select the one you forked the project to from the **Switch billing account** dropdown.

* 

1. Click **Complete order and begin installation**.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML329dc28.PNG
* Note that if you previously installed Azure Pipelines, you may need to click **grant this app access** instead.
* 

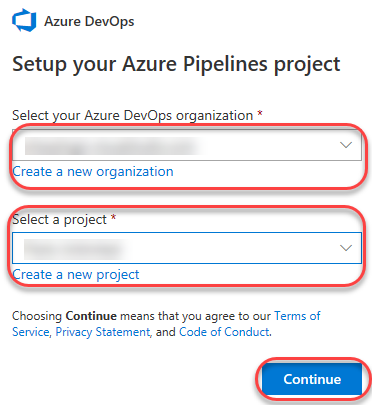
1. Select the repositories you want to include (or **All repositories**) and click **Install**.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML349c700b.PNG

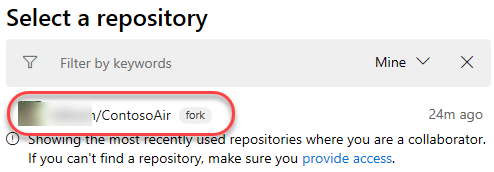
## Task 2 – Configuring an Azure Continuous Integration Pipeline

> **Talk track:** Now that Azure Pipelines has been installed in the GitHub account, we can configure Azure DevOps to use it. We created an empty Azure DevOps project ahead of time to hold and run the pipelines we need for continuous integration and continuous delivery. The first thing we’ll do is to create the build pipeline.

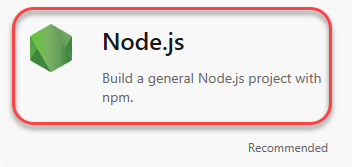
1. Select the organization and Azure DevOps project created ahead of time.

* 

1. Select the forked repo.

* 
* > **Talk track:** Every build pipeline is simply a set of tasks. Whether it’s copying files, compiling source, or publishing artifacts, the existing library of tasks covers the vast majority of scenarios. You can even create your own if you have specialized needs not already covered. We’re going to use YAML, a markup syntax that lends itself well to describing the build pipeline. Note that the Node.js pipeline as a starting point based on an analysis of our source project. We’ll replace the contents with the final YAML required for our project.

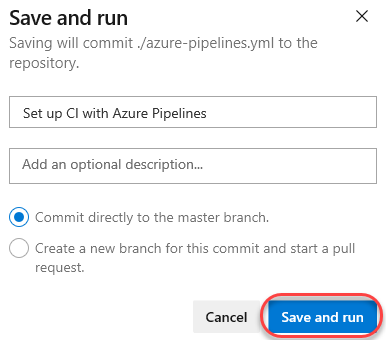
1. Select the recommended template.

* 

1. Replace the default template with the YAML below.
   1. resources:
   * - repo: self
   * queue:
   * name: Hosted VS2017
   * demands: npm
   * steps:
   * - task: CopyFiles@2
   * displayName: 'Copy Files to: $(build.artifactstagingdirectory)/Templates'
   * inputs:
   * SourceFolder: deployment
   * Contents: '\*.json'
   * TargetFolder: '$(build.artifactstagingdirectory)/Templates'
   * - task: Npm@1
   * displayName: 'npm custom'
   * inputs:
   * command: custom
   * verbose: false
   * customCommand: 'install --production'
   * - task: ArchiveFiles@2
   * displayName: 'Archive $(Build.SourcesDirectory)'
   * inputs:
   * rootFolderOrFile: '$(Build.SourcesDirectory)'
   * includeRootFolder: false
   * - task: PublishBuildArtifacts@1
   * displayName: 'Publish Artifact: drop'
2. Click **Save and run**.

* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe600c49.PNG
* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe600c49.PNG

1. Confirm the **Save and run** to commit the YAML definition directly to the master branch of the repo.

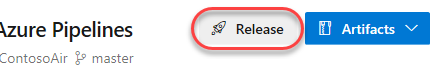
* 

1. Follow the build through to completion.

## Task 3 – Configuring an Azure Continuous Delivery Pipeline

> **Talk track:** Now that the build pipeline has been created and the first build has completed, we can turn our attention to creating a release pipeline. Like the build templates, there are many packaged options available that cover common deployment scenarios, such as publishing to Azure. But to illustrate how flexible and productive the experience is, we will build this pipeline from an empty template.

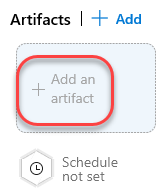
1. Click **Release**.

* 

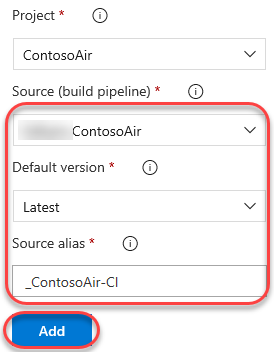
1. Click **Empty job**.

* 
* > **Talk track:** The first item to define in a release pipeline is exactly what will be released and when. In our case, it’s the output generated from the build pipeline. Note that we could also assign a schedule, such as if we wanted to release the latest build every night.

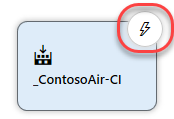
1. Click **Add an artifact**.

* 

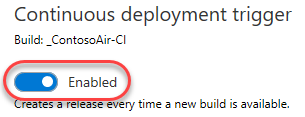
1. Set **Source** to the build pipeline created earlier and **Default version** to **Latest**. Change the **Source alias** to **“\_ContosoAir-CI”** and click **Add**.

* 
* > **Talk track:** As we did with continuous integration starting on a source commit, we also want to have this pipeline automatically start when the build pipeline completes. It’s just as easy.

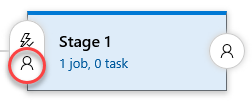
1. Click the **Triggers** button on the artifact.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML38b9ea3a.PNG

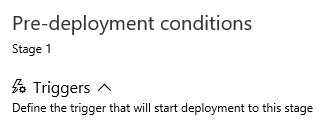
1. **Enable** continuous integration.

* 
* > **Talk track:** We also have the option of adding quality gates to the release process. For example, we could require that a specific user or group approve a release before it continues, or that they approve it after it’s been deployed. These gates provide notifications to the necessary groups, as well as polling support if you’re automating the gates using something dynamic, such as an Azure function, REST API, work item query, and more. We won’t add any of that here, but we could easily come back and do it later on.

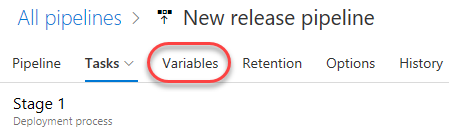
1. Click the **pre-deployment conditions** button.

* 

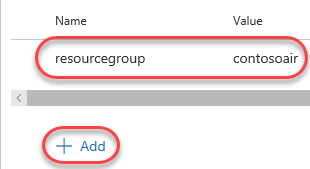
1. Review pre-deployment condition options.

* 
* > **Talk track:** In this pipeline, we’re going to need to specify the same resource group in multiple tasks, so it’s a good practice to use a pipeline variable. We’ll add one here for the new Azure resource group we want to provision our resources to. Note that there are also a variety of deployment options we can configure, as well as a retention policy.

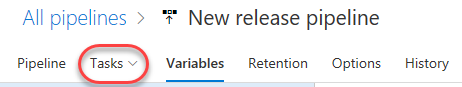
1. Select the **Variables** tab.

* 

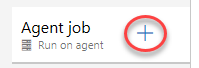
1. **Add** a **resourcegroup** variable that is not currently used by an existing resource group in your Azure account (**“contosoair”** will be used in this script).

* 
* > **Talk track:** Also, just like the build pipeline, the release pipeline is really just a set of tasks. There are many out-of-the-box tasks available, and you can build your own if needed. The first task our release requires is to set up the Azure deployment environment if it doesn’t yet exist. After we add the task, I can authorize access to the Azure account I want to deploy to and instruct it to use the variable name we just specified for the resource group name.

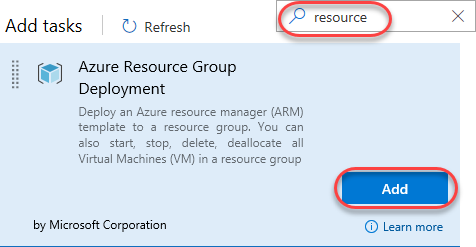
1. Select the **Tasks** tab.

* 

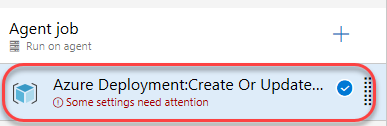
1. Click the **Add task** button.

* 

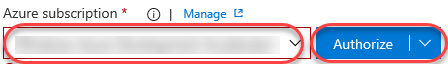
1. Search for **“resource”** and **Add** an **Azure Resource Group Deployment** task.

* 

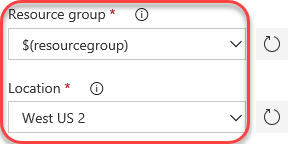
1. Select the newly created task.

* 

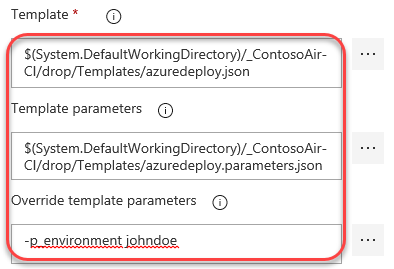
1. Select and authorize an Azure subscription.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML393f8328.PNG

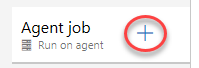
1. Set the **Resource group** to **“$(resourcegroup)”** and select a **Location**.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML42d91670.PNG
* > **Talk track:** Rather than having to manually create the Azure resources required to host the web app, the team has defined an Azure Resource Manager—or ARM—template that describes the environment in JSON. This allows the environment definition to be updated and managed like any other source file. These were the files we copied to the Templates folder during the build pipeline. You can also override the template parameters as part of this configuration, which we’ll do here using my name.

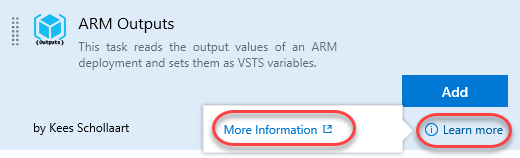
1. Enter the settings below. You can use the browse navigation to select them from the most recent build output.

* Template: **$(System.DefaultWorkingDirectory)/\_ContosoAir-CI/drop/Templates/azuredeploy.json** Template parameters: **$(System.DefaultWorkingDirectory)/\_ContosoAir-CI/drop/Templates/azuredeploy.parameters.json**
* You will also need to set **Override template parameters** to generate an Azure app service name that is globally unique, so your name is recommended. For example, if your name is **John Doe**, use something like **“-p\_environment johndoe”**. This will be used as part of the app service name in Azure, so please limit it to supported characters.
* 
* > **Talk track:** When this task completes, it will have generated an Azure resource group with the resources required to run our application. However, the ARM template does some processing of the variables to generate names for the resources based on the input variables, which we will want to use in future tasks. While we could potentially hardcode those variables, it could introduce problems if changes are made in the future, so we’ll use the ARM Outputs task to retrieve those values and put them into pipeline variables for us to use. This task happens to be a 3rd party task I installed earlier from the Visual Studio Marketplace. It contains this and many other extensions for Azure DevOps from both Microsoft and 3rd parties.

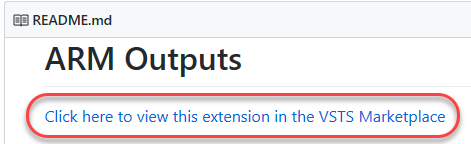
1. Click the **Add task** button.

* 

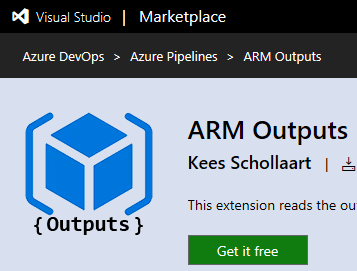
1. Search for **“arm”** and select **Learn more | More information**. This will open the GitHub project for this extension in a new tab.

* 

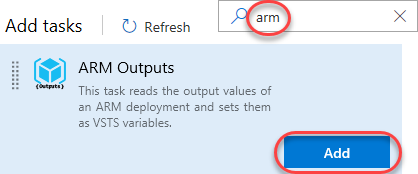
1. Click the link to the Visual Studio Marketplace.

* 

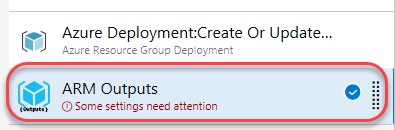
1. Close the new tab.

* 
* > **Talk track:** Now let’s get back to adding the ARM Outputs task. The key variable we care about here is the name of the app service created, which our ARM template has specified as an output. This task will populate it for us to use as the “web” variable in the next task.

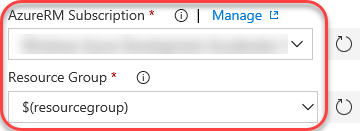
1. **Add** an **ARM Outputs** task.

* 

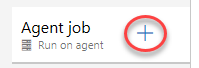
1. Select the newly created task.

* 

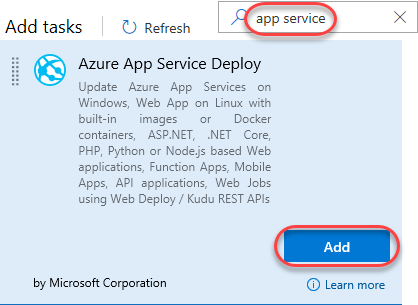
1. Select the same subscription from the previous task and enter the same resource group variable name.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML42ecfcec.PNG
* > **Talk track:** Finally, we can deploy the app service. We’ll use the same subscription as earlier and specify the web variable as the name of the app service we want to deploy to. By this time in the pipeline, it will have been filled in for us by the ARM Outputs task. Also note that we have the option to specify a slot to deploy to, but we’ll talk about deployment slots later on.

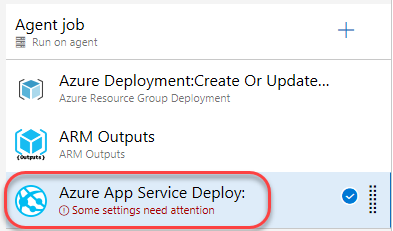
1. Click the **Add task** button.

* 

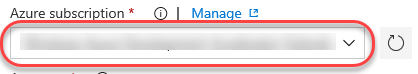
1. Search for **“app service”** and **Add** an **Azure App Service Deploy** task.

* 

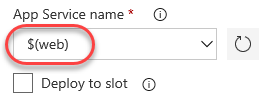
1. Select the newly created task.

* 

1. Select the same subscription as earlier.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML38dcffeb.PNG

1. Enter the **App Service name** of **“$(web)”**.

* 

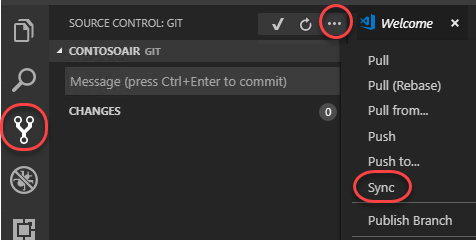
1. **Save** the pipeline.

* 

## Task 4 – Invoking Continuous Delivery from GitHub to Azure

> **Talk track:** Now that we have our pipelines in place, it’s time to commit a change to the master branch on GitHub. We’re going to pull down the azure-pipelines.yml file added by Azure DevOps during the build creation and commit a slight edit to trigger the CI/CD process.

1. Open the ContosoAir project in Visual Studio Code.
2. From the **Source Control** tab, select **Sync** from the **More Actions** dropdown.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML1364534f.PNG

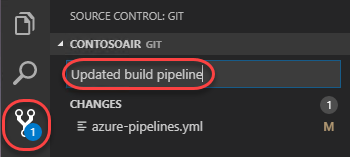
1. From the **Explorer** tab, open **azure-pipelines.yml**.

* 
* > **Talk track:** Before we make our change, let’s take a quick look at the build tasks. There are four steps required for the build. First, deployment templates are copied to a target folder for use during the release process. Next, the project is built with NPM. After that, the built solution is archived and finally published for the release pipeline to access. With the Azure Pipelines extension for Visual Studio Code, you get a great YAML editing experience, including support for IntelliSense.

1. Examine the tasks within the pipeline definition.
2. Make a change to show IntelliSense offered by the extension.
3. Undo any changes to avoid breaking the definition.
4. Add a newline to the end of the file. The purpose is simply to produce a change to the file you can commit, but without impacting the working build.
5. Press **Ctrl+S** to save the file.

* > **Talk track:** Now we can commit and push the updated build definition to GitHub. This will invoke a continuous integration build in Azure DevOps, which will trigger a continuous delivery to Azure upon completion.

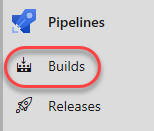
1. From the **Source Control** tab, enter a commit message like **“Updated build pipeline”** and press **Ctrl+Enter** to commit. Confirm if prompted.

* 

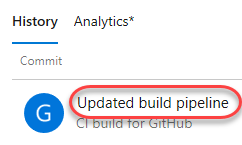
1. Press the **Synchronize Changes** button at the bottom of the window to push the commit to the server. Confirm if prompted.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML136c0f0b.PNG
* > **Talk track:** Back in Azure DevOps, we can see that our build pipeline has kicked off a new build. We can follow as it executes the tasks we defined earlier, and even get a real-time view into what’s going on at each step. When the build completes, we can review the logs and any tests that were performed as part of the process.

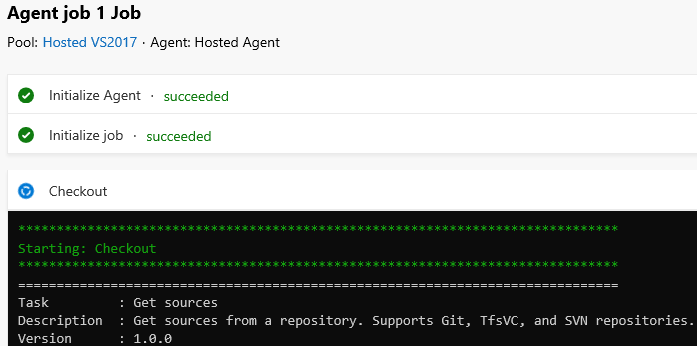
1. Return to Azure DevOps and navigate to the **Builds** hub.

* 

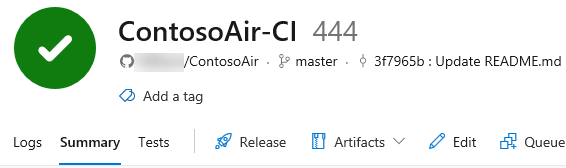
1. Click the new build.

* 

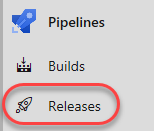
1. Track the build tasks.

* 

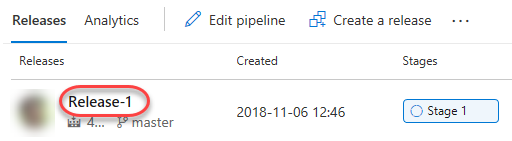
1. Follow the build through to completion.

* 
* > **Talk track:** Now that the build has completed, let’s check out the release. It was automatically invoked by the successful completion of the build pipeline, and we can follow it all the same. Since this is the first time we’re deploying, Azure will need to provision the resources. That can take a minute, so let’s check back in later.

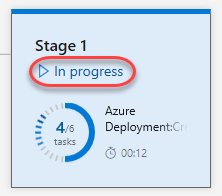
1. Navigate to the **Releases** hub.

* 

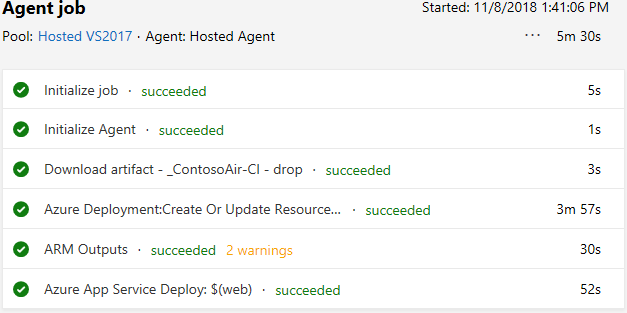
1. Select the new release. If one is not immediately available, click the **Refresh** option.

* 

1. Click **In progress** to follow the release process.

* 

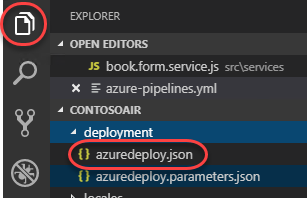
1. Note that it will take a few minutes (around 5 at the time of drafting) for the app to finish deploying due to heavy first-time operations. Move ahead to the next step while it works in the backgroud.

* 

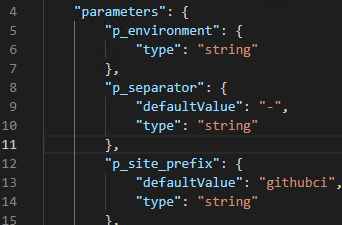
## Task 5 – Reviewing the ARM template

> **Talk track:** A lot of what’s going on now was directed by the ARM template the DevOps team put together. Let’s take a tour through it to understand how it’s structured.

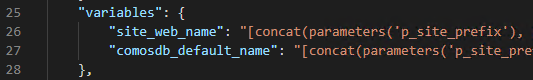
1. Return to **Visual Studio Code**.
2. From the **Explorer** tab, open **/deployment/azuredeploy.json**.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML13deea1f.PNG
* > **Talk track:** The first section defines the parameters the template expects. They can all be overridden externally by a parameter file, and, in our case, via the pipeline editor. Note that they all have default values, except for the environment parameter, which must be supplied.

1. Review **parameters**.

* 
* > **Talk track:** The variables section defines a set of template variables that can be used to reduce complex expressions into an easily manageable term. For example, you can concatenate a combination of variables, resource properties, and other accessible functions to generate names or parameters for other parts of the template. In this case, the app service and Cosmos DB have their names generated from the set of variables passed in. Note the site\_web\_name variable, which is the name of the app service that will be created and the string the release pipeline needs to know for deployment.

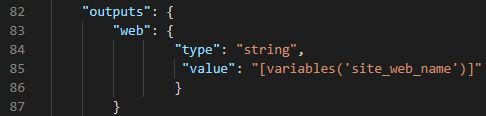
1. Review **variables**.

* 
* > **Talk track:** The resources section defines the actual resources to be created in Azure. This section can get pretty complex and offers incredible flexibility for defining what your application needs, how they are to be configured, and what dependencies they have on each other. This template only explicitly defines two resources: the Cosmos DB and the app service.

1. Review **resources**.

* 
* > **Talk track:** The outputs section provides a way to expose data from this template for external services. This is the part that the ARM Outputs task will use to populate the web pipeline variable with the complex site\_web\_name template variable.

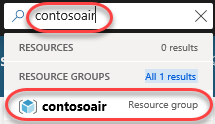
1. Review **outputs**.

* 

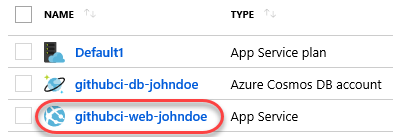
## Task 6 – Reviewing the Application and Azure Portal

> **Talk track:** While the resources finish spinning up, let’s take a quick tour of the Azure portal. It’s thoughtfully designed and easy to use. The navigation on the left hand side provides access to major platform components, such as app services and virtual machines. Solutions are organized as resource groups, which are logical collections of the resources used to run your solution. We’ll search for the solution our release pipeline created earlier.

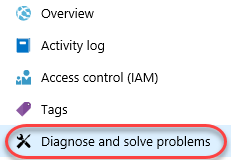
1. Switch to the Aure portal tab (<https://portal.azure.com>).
2. Review the left-hand navigation.
3. Search for your resource group name and open it.

* 
* > **Talk track:** Our solution requires three resources in Azure. The app service is our web site and the app service plan is the virtual server farm the site is deployed to. The Cosmos DB is a globally-distributed, multi-model database service instance we’re using to manage all of the data in the site. Let’s take a closer look at the app service.

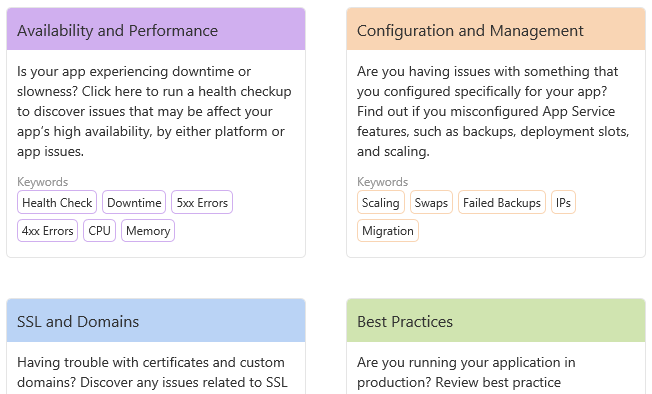
1. Click the **App Service**. If this isn’t available yet, check in on the release pipeline to make sure the Azure Deployment task didn’t fail. Otherwise, refresh occassionaly until it is. Keep in mind that the app service will be available before the app itself is deployed, so the actual site itself won’t be ready until the pipeline completes.

* 
* > **Talk track:** This view is the dashboard for the app service and provides convenient access to virtually everything we could ever need to do. One of the most useful features for DevOps professionals is the “diagnose and solve problems” view that offers quick access to a variety of self-diagnostic features. These features include checks for typical issues related to availability, performance, and configuration.

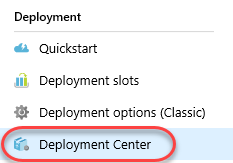
1. Click **Diagnose and solve problems**.

* 

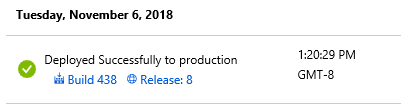
1. Review the options.

* 
* > **Talk track:** The deployment center provides a single place to track deployment, such as those that are automated via pipeline. You can also use and track deployment slots, which allow you to have additional release targets. For example, you might make it a policy to always deploy to the staging slot so that your team has an opportunity to review and run tests before making it public. Configuring a release pipeline to push to a specific slot is a setting available in deployment tasks.

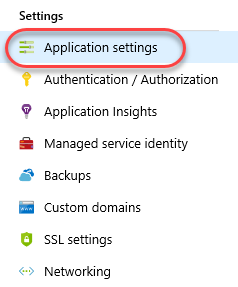
1. Click **Deployment Center**.

* 

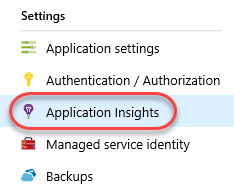
1. Review the deployment.

* 
* > **Talk track:** The “application settings” view enables you to define system-level settings for the environment, such as versions of .NET or PHP. You can also configure virtual applications and directories, as well as application-level settings, such as connection strings.

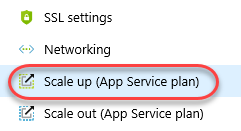
1. Click **Application settings**.

* 
* > **Talk track:** Application Insights is one of the most valuable services for DevOps teams. It provides performance tracking and management features for every level of an application. We haven’t configured it for this project yet, but once it’s in place, you can trace an action in a web browser all the way through the web request into an API and down to the resources it’s dependent on. If there’s an error somewhere along the way, it’s really easy to diagnose the cause so that teams spend more time working on improvements than troubleshooting.

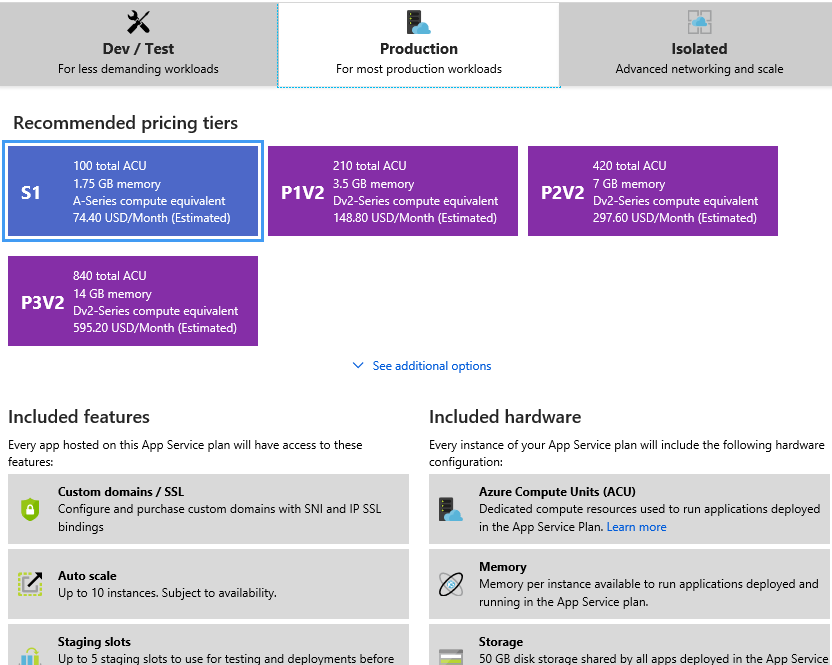
1. Click **Application Insights**.

* 
* > **Talk track:** One of the great benefits of a cloud platform is how easily you can scale a platform up and out. For example, we have the option here to scale our current application up to use a pretty powerful set of virtual hardware. Or, depending on our needs, we can use the scale out option to add more instances as well. There is even an autoscale option to automatically add and remove instances based on load.

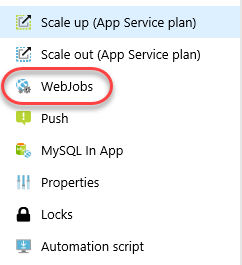
1. Click **Scale up**.

* 

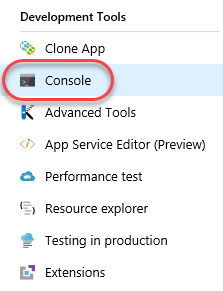
1. Review options.

* 
* > **Talk track:** In addition to web sites, you can also build and deploy web jobs. These are standalone apps or scripts that can be invoked via web hook. You can also use them in combination with a scheduler to perform regular tasks, such as batch updates to your data store.

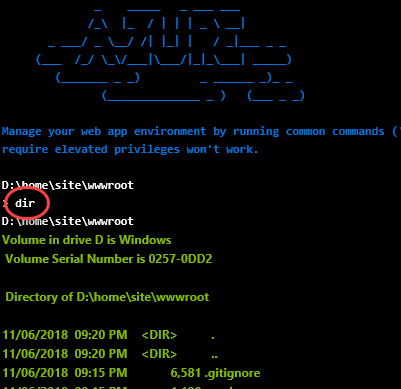
1. Click **WebJobs**.

* 
* > **Talk track:** There’s also a console option for you to explore what’s going on in your service. For example, let’s get a directory listing of the web root.

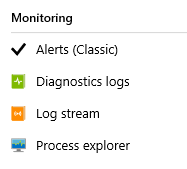
1. Click **Console**.

* 

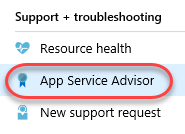
1. Execute a **“dir”** command.

* 
* > **Talk track:** There are also plenty of built in monitoring and alerting features. These save you a lot of time so you can focus on developing business value instead. And if you’re looking for advice on places to improve the application, there’s the App Service Advisor. Things are looking good now, but we’ll want to keep an eye on this for future suggestions.

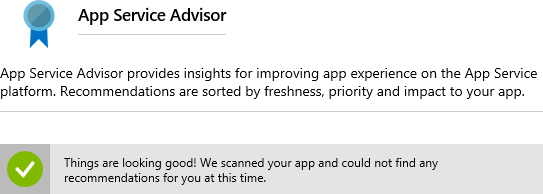
1. Review monitoring options.

* 

1. Click **App Service Advisor**.

* 

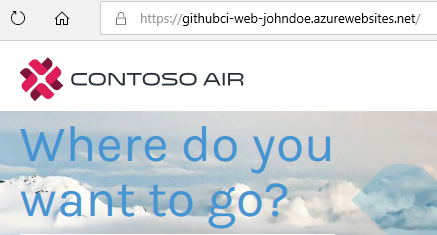
1. Review insights.

* 
* > **Talk track:** That was a quick tour of the app service configuration, so let’s check out the actual site in the cloud. We just took a project in GitHub and set up a sophisticated, automated deployment to Azure in mere minutes!

1. Click the **URL** to open the site.

* C:\Users\Ed\AppData\Local\Temp\SNAGHTML39096e7d.PNG
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML39096e7d.PNG

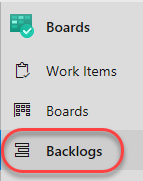
1. Review the site. Keep the browser window open for later.

* 

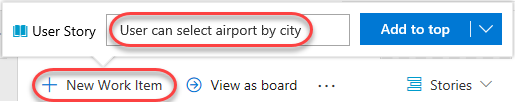
## Task 7 – Managing GitHub Projects with Azure DevOps

> **Talk track:** Azure DevOps provides a wealth of project management functionality that spans Kanban boards, backlogs, team dashboards, and custom reporting. By connecting Azure Boards with GitHub repositories, you can create links between GitHub commits and pull requests to work items tracked in Azure Boards. This enables a seamless way for you to use GitHub for software development while using Azure Boards to plan and track your work.

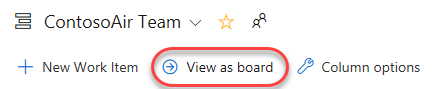
1. Return to the Azure DevOps tab.
2. Navigate to **Boards | Backlogs**.

* 
* > **Talk track:** In our scenario, users will need to be able to book flights by selecting the cities involved. We will create a new user story to sort the airports listed in the booking form in alphabetical order by city. Ordinarily we would create the user story at a higher level and add tasks to define how the story is to be implemented, but for our demo purposes here we’ll leave it as a single work item.

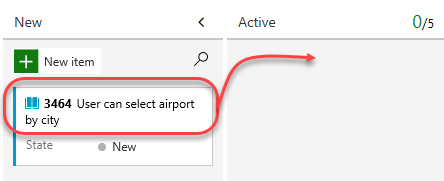
1. Click **New Work Item** and add a user story with the title **“User can select airport by city”**. Press **Enter** to create.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML9701286.PNG
* > **Talk track:** In addition to working with work items in a backlog, we have a very flexible Kanban board option. With the board, we can edit items on a card in line, or even drag cards around to change their state and assignment. Let’s take ownership of the new user story so we can begin work.

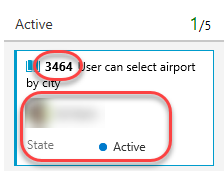
1. Click **View as board**.

* 

1. Drag the newly created user story to the **Active** column.

* 

1. Dropping the user story onto the **Active** column assigns it to you and sets its **State** to **Active**. Make note of the task ID for reference later during a future commit and pull request.

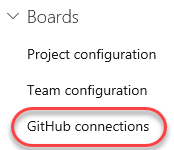
* 

> **Talk track:** In order to complete our integration, we’ll need to wire up a connection between this project and the GitHub repo.

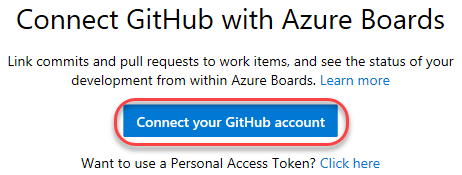
1. Click **Project settings**.

* 

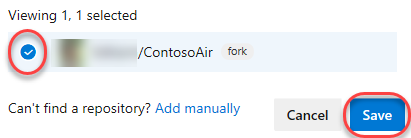
1. Under **Boards**, select **GitHub connections**.

* 

1. Click **Connect your GitHub account**.

* 

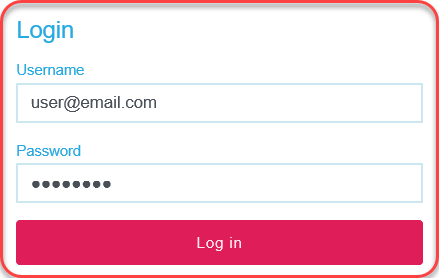
1. Select the project repo and click **Save**.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML2712de8e.PNG
* > **Talk track:** Let’s take a look at our deployed site to see what the current booking experience is like. As you can see, the airports appear to be sorted by airport code, which isn’t the behavior we want our users to see.

1. Return to the web app tab and click **Login**.

* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe8868fc.PNG
* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe8868fc.PNG

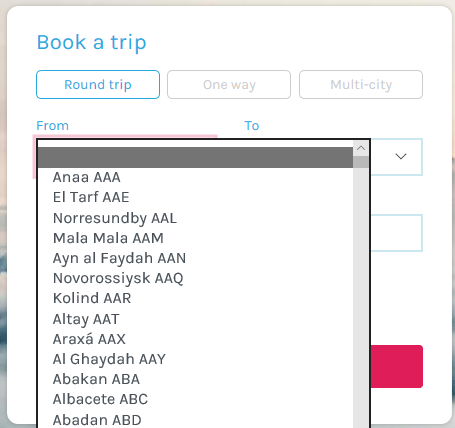
1. Log in with any email and password.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe88013a.PNG

1. Click **Book**.

* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe876e60.PNG
* C:\Users\Ed\AppData\Local\Temp\SNAGHTMLe876e60.PNG

1. Expand the airport dropdown to note that it’s not sorted alphabetically by city.

* 

## Task 8 – Committing to Complete a Task

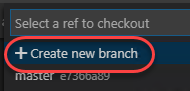
1. Return to **Visual Studio Code**.

* > **Talk track:** We’ll start off by creating a new branch for this task. The work itself is pretty straightforward. We just need to locate the place where airports are provided to the user experience and make sure they’re being sorted by city name.

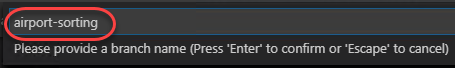
1. Click the **master** branch at the bottom of the window.

* C:\Users\Ed\AppData\Local\Temp\SNAGHTML137df3cb.PNG
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML137df3cb.PNG

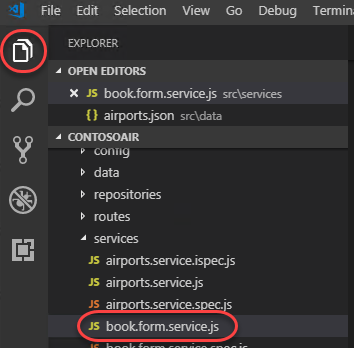
1. From the top of the screen, click **Create new branch**.

* 

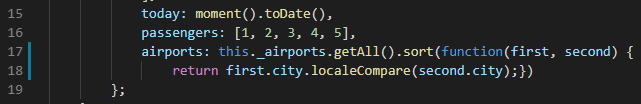
1. Enter the name **“airport-sorting”** and press **Enter**. This will activate the new branch.

* 

1. From the **Explorer** tab, open **src/services/book.form.service.js**.

* 

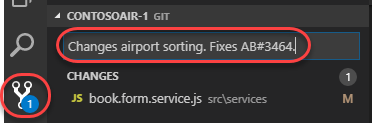
1. Locate the **getForm** function and replace the existing **airports** initializer with the code below. This will sort the airports by city.

* airports: this.\_airports.getAll().sort(function(first, second) {
* return first.city.localeCompare(second.city);})
* 

1. Press **Ctrl+S** to save the file.

* > **Talk track:** We’ll skip testing this locally for the sake of the demo. Instead, we’ll commit it using a comment that includes special syntax to link it to the Azure Boards task we saw earlier. Now this commit will become trackable from project management, as long as we include the phrase “Fixes AB#ID”.

1. Switch to the **Source Control** tab and enter a commit message of **“Changes airport sorting. Fixes AB#3464.”**, but replace **3464** with the actual ID of the Azure Boards task. Press **Ctrl+Enter** and confirm the commit if prompted.

* 
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML97d87a7.PNG

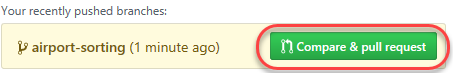
1. Click the **Publish Changes** button at the bottom of the screen.

* C:\Users\Ed\AppData\Local\Temp\SNAGHTML13836058.PNG
* C:\Users\Ed\AppData\Local\Temp\SNAGHTML13836058.PNG

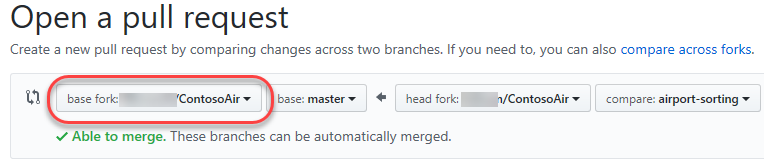
1. When the push has completed, return to the GitHub browser tab.

* > **Talk track:** With the commit pushed, we’ll create a pull request to drive those changes back into the master branch. In this case we’re inheriting the title from the commit, but having the pull request mention “Fixes AB#ID” will link and complete the target work item when the pull request is merged.

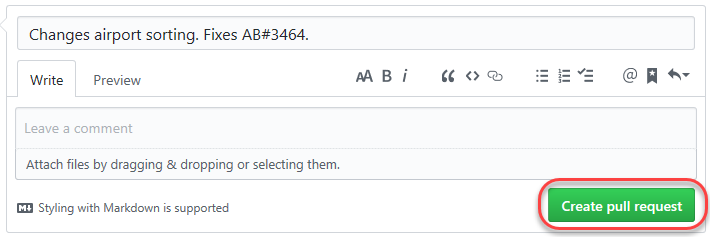
1. Click **Compare & pull request**, which should appear on its own. If not, refresh.

* 

1. Change the **base fork** to point at your project. By default it points at the original Microsoft repo, so be sure to change it.

* 

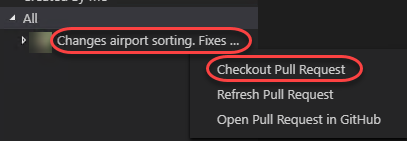
1. The title should initialize to the commit message entered earlier. Click **Create pull request**.

* 

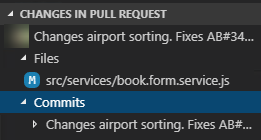
1. Return to Visual Studio Code.

* > **Talk track:** Now we’ll switch to the other side of the pull request and take on the role of reviewer. We can use Visual Studio Code to check out the pull request, analyze changes, and comment. Assuming we trust the fix, we can merge the pull request to update master and kick off the CI/CD.

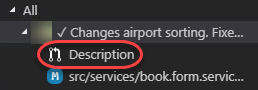
1. Under **GitHub Pull Requests | All**, right-click the pull request and select **Checkout Pull Request**.

* 

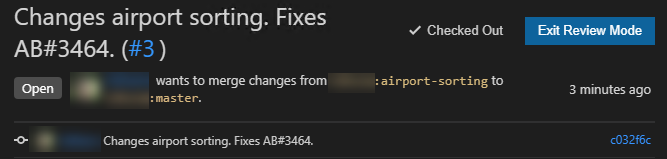
1. Expand the **Changes in Pull Request** tree.

* 

1. Select the **Description** from under the original pull request.

* 

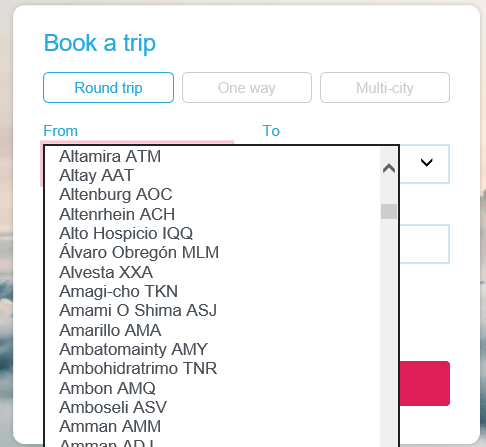
1. Review the details of the pull request.

* 

1. Click **Merge pull request** and confirm the merge.

* 
* > **Talk track:** Once the deployment works its way through build and release, we can confirm the new functionality.

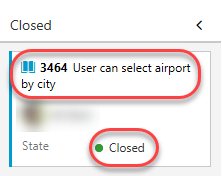
1. Follow the CI/CD pipeline through to completion.
2. Refresh the web app site. Return to the booking page (you’ll need to log in again) and confirm the airports are sorted by city now (scroll down past the airports with no city name).

* 

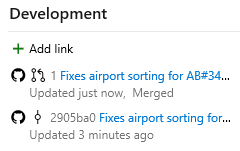
1. Return to the Azure DevOps tab open to the Kanban board.

* > **Talk track:** Since the user story we were working on was linked in a pull request that was approved, Azure DevOps will automatically transition the state of the work item to “Closed”. You can also see that the related GitHub commits and pull request were linked to the work item.

1. The user story should have already moved to the **Closed** state and column. Click to open it.

* 

1. The commit and pull request should now be visible under **Development**.

* 

# Summary

Many organizations have their projects hosted in GitHub, and we just showed how you can set up automated deployment to Azure in minutes. And it doesn’t matter what kind of application they’re building or what kind of environment they’re deploying to. Once this automation is in place, companies can turn their focus to developing business value instead of infrastructure.