Azure E-book Series

Microsoft

Building cloud-native applications with Node.js and Azure

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

This e-book presents a structured approach for building Node.js cloud-native applications

01 /
The basics of cloud-native applications and Node.js

02 /
The benefits of cloud-native applications for Node.js

03 /
Principles for architecting your cloud-native application with Node.js

04 /
How to build a cloud-native application with Node.js and Azure

Node.js and cloud-native applications are changing the way developers build

Instead of monoliths, applications are decomposed into smaller, decentralized services

These services communicate through APIs or by using asynchronous messaging. Applications scale horizontally, adding new instances as demand requires.

Node.js on the cloud
/ single threaded
/ event driven
/ increased execution speed
/ designed for elastic scale
automated self-management

Traditional web servers
/ multi-threaded
/ always-on
/ occassional bug updates
/ manual management
/ monolithic, centralized

These trends bring new challenges. Application state is distributed. Operations are done in parallel and asynchronously. The system as a whole must be resilient when failures occur. Deployments must be automated and predictable. Monitoring and telemetry are critical for gaining insight into the system.

Chapter 1 /

The basics of cloud-native applications and Node.js

Cloud-native applications are ones that are designed specifically for a cloud computing architecture. These applications are designed to take advantage of cloud computing frameworks, which are composed of loosely coupled cloud services. Such applications are ones that have a high degree of uptime and are highly available, from wherever there's an internet connection. These applications are extremely scalable.

In the simplest way to describe it, Node.js is server-side JavaScript. It's non-blocking and event-driven; thanks to its event loop, it is optimized for handling asynchronous I/O, such as calls to databases or external services, two very common operations in modern web apps. And, because it's JavaScript, it's fast and flexible to use with other types of code. JavaScript was originally designed as a language only for client-side web applications, but Node.js allows for server-side development, making it one of the most popular programming languages. Node.js allows web developers to expand their creative potential by creating servers, command-line tools, and desktop applications. Using asynchronous I/O, the server can do more than one thing at a time, a key requirement for real-time apps like chat, games and live statistics.

This makes Node.js a natural for the cloud, with its anywhere access and need for speed. With Azure, which is built to support many architectures (including microservices), databases, and services, as well as having support for containers, Node.js is all the more powerful. For example, on Azure it's easy to add identity and access management through Azure Active Directory, vision and speech capabilities with Azure Cognitive Services, or data insights through Azure Data Lake Analytics – just to name a few of the many examples.

Chapter 2 /

The benefits of cloud native for Node.js development

Application development and IT system management is undergoing revolutionary change driven by the cloud. Fast, agile, inexpensive, and massively scalable infrastructure is improving operational efficiency and enabling faster-time-to-value across industries. The emergence of containers, with their fast startup, standardized application packaging, and isolation model, is further contributing to efficiency and agility.

There are many ways businesses can take advantage of the capabilities this brings. Quick and reliable data processing, error analysis, and quick code deployment can enable a startup to grow faster than larger companies. You can reduce build times and user customization enabled by using Node.js when streaming content. Companies can also align dispersed development teams onto a single language with Node.js, which means significantly less time is spent writing code. Azure cloud services provide Node.js developers the ability to quickly add new functionality to their software while remaining stable and secure in production. By using Azure cloud services, you can develop, package, deploy and scale powerful applications on many of platforms.

Pillars of cloud-native applications with Node.js

As more and more applications are being developed in the cloud, agile practices have emerged. Agile development promotes small, well planned, iterations by highly collaborative teams, resulting in continuous delivery. The pillars of agile methodology likely fit well with why you chose to work with Node.js in the first place.

Speed
Deliver the responsiveness users demand. Because Node.js runs on the Chrome V8 engine, which is optimized for speed, and because of its asynchronous processing, Node.js apps are very fast. With Node.js, speed is also about development agility, which using JavaScript enables. Lastly, with around half a million modules (and counting) on NPM, the Node.js Package Manager, developers can easily integrate libraries for many common tasks.

Flexibility with components
Azure supports many architectures, operating systems, tools, services, and databases. Because of this, Node.js developers can connect to the functionality they need to support their applications: databases, identity providers, cognitive services, etc.

Reliability
Azure offers a large variety of hosting options, including Linux and Kubernetes. With containers like Docker you can develop new features and functionality within the production environment, but isolate them so that you can test and roll out new features or fixes with confidence.

Some of the greatest benefits of working in the cloud are the monitoring and analytics capabilities. By tracking costs and efficiencies of your work, you can understand what is working well, and make better decisions. This will also allow you to simplify your environment and free up resources.

Chapter 3 /

Principles for architecting your cloud-native application

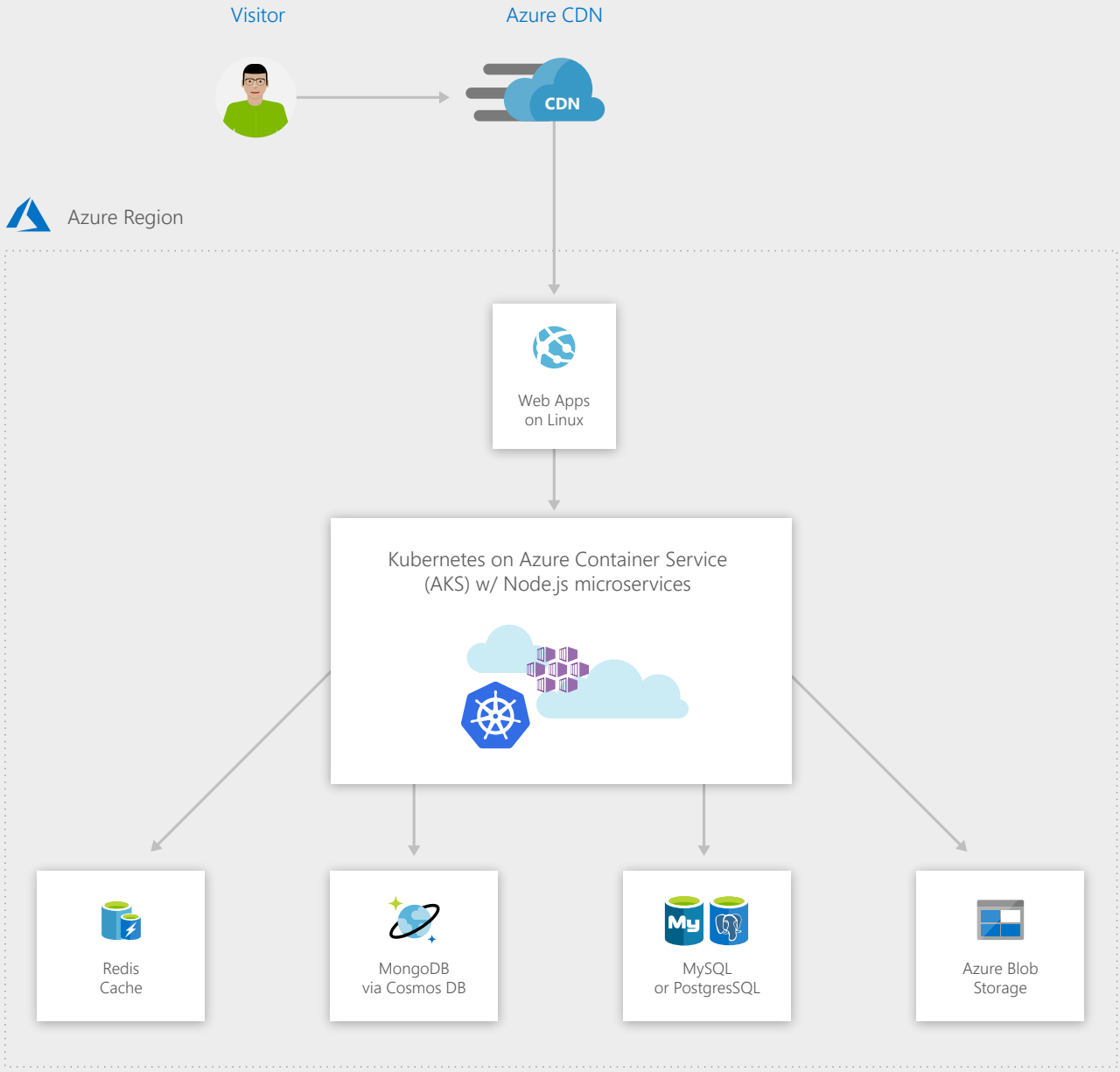
In this section, we will walk through the process of creating, hosting, and architecting your cloud-native application. The architecture in this chapter includes the core components available to you as a Node.js developer using Azure to build fast, reliable, and flexible applications in the cloud. With the cloud you get to pick and choose what services you use to build your application. Depending on your business logic you will probably use some combination of the following; Web Apps on Linux, Kubernetes, CosmosDB (MongoDB), MySQL/PostgreSQL, Redis Cache, Blob storage.

With the cloud you get to pick and choose what services you use to build your application

Web apps on Linux
Modern cloud-native applications usually separate the web-based frontend from the backend, which often provides RESTful APIs and/or GraphQL. Web Apps on Linux is a Platform-as-a-Service offering on Azure that allows serving web applications from a completely managed environment. Web Apps on Linux supports Node.js apps and static websites (as well as many other technologies), and it supports autoscaling, Continuous Integration and Continuous Delivery, and more.

Kubernetes on Azure Container Service (AKS)
Azure Container Service (AKS) offers a fully-managed Kubernetes cluster. You can pick and choose the number and kind of worker nodes you want, and Azure takes care of provisioning, managing and updating them, giving you access to a full Kubernetes cluster that can be interacted with using the web UI or kubectl.

Cosmos DB (MongoDB)
Cosmos DB is a managed, massively scalable, NoSQL Database-as-a-Service available on Azure, that offers full protocol compatibility with MongoDB. Applications that are designed for MongoDB can connect to Cosmos DB using the same libraries and tools. Cosmos DB supports multiple performance tiers, and allows geo-replication with selectable consistency levels.



Azure DB for MySQL and PostgreSQL
In addition to installing them on a Virtual Machine, with Azure you can run MySQL or PostgreSQL, two popular relational databases, as fully-managed services too.

Redis Cache
Azure Redis Cache offers the popular Redis in-memory key-value storage as a fully managed service.

Blob Storage
Lastly, Azure Blob Storage is a massively scalable object storage. Your application can use it to store static assets, images, cached data, and anything unstructured, in a simple and cost-effective way. Now that we have introduced the components you will be working with, let's get familiar with how to build and deploy a simple Node.js web app in Azure.

Before you get started

To complete the examples in this e-book, you'll need the following:

- Install Git
- Install Node.js
- An Azure free account

You will also need the Azure CLI 2.0, which is Azure's new command line experience for managing Azure resources. You can use it in your browser with Azure Cloud Shell, or you can install it on macOS, Linux, and Windows and run it from the command line. After you've started Azure Cloud Shell or installed the Azure CLI, see Get Started with Azure CLI 2.0.

Chapter 4 /

Build the app

01 /
Create a Node.js web app in Azure App Service on Linux

02 /
Deploy an Azure Container Service (AKS) cluster

03 /
Azure Cosmos DB: Migrate an existing Node.js Mongo DB web app

04 /
Azure Database for MySQL: Use Node.js to connect and query data

05 /
How to use Azure Redis Cache with Node.js

06 /
How to use Blob storage from Node.js

Create a Node.js web app in Azure App Service on Linux

In this section, we'll create a Node.js web app in Azure App Service on Linux using a built-in image, using Azure CLI. We'll use Git to deploy the Node.js code to the web app.

Web Apps is a service for hosting web applications, REST APIs, and mobile back ends. App Service on Linux provides a highly scalable, self-patching web hosting service using the Linux operating system. App Service on Linux supports Node.js as a built-in image, as well as PHP, .NET Core, Ruby, and more to increase developer productivity.

Customers can use App Service on Linux to host web apps natively on Linux for supported application stacks.

Before you begin

Install the Azure CLI 2.0 software as described here and sign up for an Azure free account. You'll also need to download the sample code here below.

In a terminal window on your machine, clone the sample app repository to your local machine by running the following command.

```
git clone https://github.com/Azure-Samples/nodejs-docs-hello-world
```

Use this terminal window to run all the commands in this example.
Change to the directory that contains the sample code.

```
cd nodejs-docs-hello-world
```

Get set up

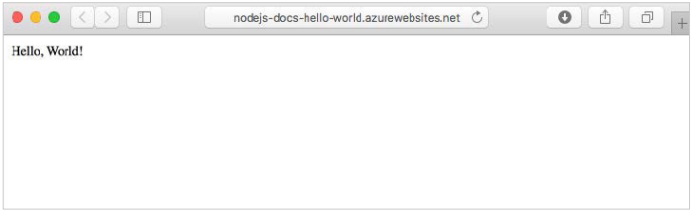
Now that you have what you need, there are a few tasks to do before you build your app.

Run the app locally
Run the application locally by opening a terminal window and using the `npm start` script to launch the built in Node.js HTTP server.

```
npm start
```

Open a web browser, and navigate to the sample app at `http://localhost:1337`.

You'll see the **Hello World** message from the sample app displayed in the page.

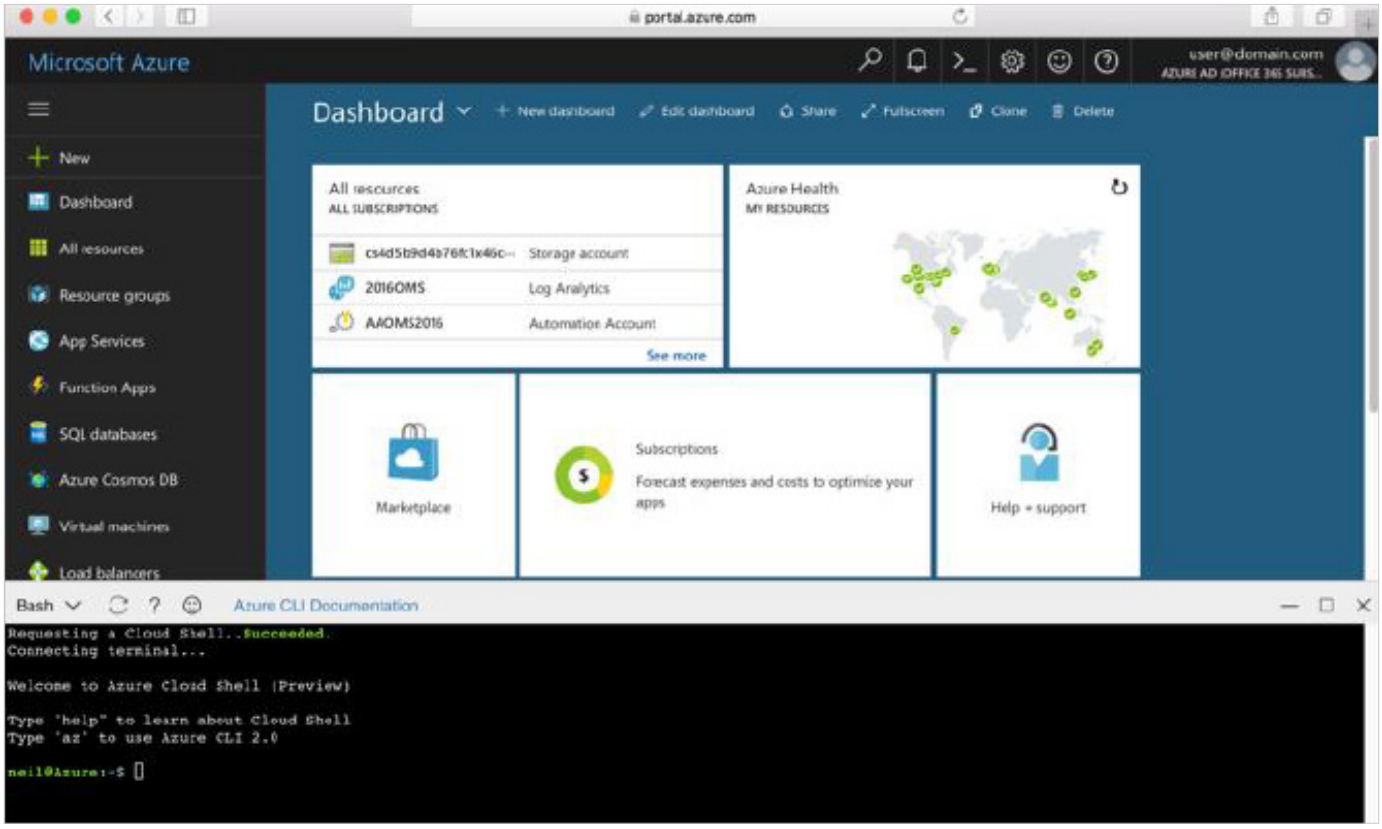


In your terminal window, press **Ctrl+C** to exit the web server.

Launch Azure Cloud Shell

The Azure Cloud Shell is a free Bash shell that you can run directly within the Azure portal. It has the Azure CLI preinstalled and configured to use with your account. Click the Cloud Shell button on the menu in the upper-right of the Azure portal.

This button launches an interactive shell that you can use to run the steps in this topic:



Create a deployment user

In the Cloud Shell, create deployment credentials with the az webapp deployment user set command.

A deployment user is required for FTP and local Git deployment to a web app. The user name and password are account level. *They are different from your Azure subscription credentials.*

In the following command, replace <username> and <password> with a new user name and password. The user name must be unique. The password must be at least eight characters long, with two of the following three elements: letters, numbers, symbols.

```
az webapp deployment user set --user-name <username> --password <password>
```

You might get an error at this point. Use the following guidance:

If you get a `'Conflict'. Details: 409` error, change the username.

If you get a `'Bad Request'. Details: 400` error, use a stronger password.

You only need to create this deployment user once. You can use it for all your Azure deployments.

NOTE

Record the user name and password. You need them to deploy the web app later.

Create a resource group

In the Cloud Shell, create a resource group with the az group create command.

A resource group is a logical container into which Azure resources like web apps, databases, and storage accounts are deployed and managed.

The following example creates a resource group named *myResourceGroup* in the *West Europe* location.

```
az group create --name myResourceGroup --location "West Europe"
```

As a best practice, create your resource group and the resources in a region near you. To see all supported locations for App Service plans, run the `az appservice list-locations` command.

Create an Azure App Service plan

In the Cloud Shell, create an App Service plan with the az appservice plan create command.

An App Service plan specifies the location, size, and features of the web server farm that hosts your app. You can save money when hosting multiple apps by configuring the web apps to share a single App Service plan.

An App Service plan defines:

- Region (for example: North Europe, East US, or Southeast Asia)
- Instance size (e.g: small, medium, or large)
- Scale count (1 to 10 instances)
- SKU (Basic, Standard)

The following example creates an App Service plan named myAppServicePlan in the Standard pricing tier with one instance (S1), and in a Linux container:

```
az appservice plan create \
    --name myAppServicePlan \
    --resource-group myResourceGroup \
    --sku S1 \
    --is-linux
```

When you have created the App Service plan, the Azure CLI shows information similar to the following example:

```
{
  "adminSiteName": null,
  "appServicePlanName": "myAppServicePlan",
  "geoRegion": "West Europe",
  "hostingEnvironmentProfile": null,
  "id": "/subscriptions/0000-0000/resourceGroups/myResourceGroup/providers/Microsoft.Web/serverfarms/myAppServicePlan",
  "kind": "app",
  "location": "West Europe",
  "maximumNumberOfWorkers": 1,
  "name": "myAppServicePlan",
  < JSON data removed for brevity. >
  "targetWorkerSizeId": 0,
  "type": "Microsoft.Web/serverfarms",
  "workerTierName": null
}
```

Build and deploy

Now you're ready to build and deploy your app. Follow these steps.

Create a web app with built-in image

In the Cloud Shell, create a web app in the `myAppServicePlan` App Service plan with the az webapp create command. Don't forget to replace `<app_name>` with a unique app name.

The runtime in the following command is set to `NODE|6.9.` To see all supported runtimes, run az webapp list-runtimes.

```
az webapp create \
  --resource-group myResourceGroup \
  --plan myAppServicePlan \
  --name <app_name> \
  --runtime "NODE|6.9" \
  --deployment-local-git
```

When you have created the web app, the Azure CLI shows output similar to the following example:

```
Local git is configured with url of 'https://<username>@<app_name>.scm.azurewebsites.net/<app_name>.git'
{
  "availabilityState": "Normal",
  "clientAffinityEnabled": true,
  "clientCertEnabled": false,
  "cloningInfo": null,
  "containerSize": 0,
  "dailyMemoryTimeQuota": 0,
  "defaultHostName": "<app_name>.azurewebsites.net",
  "deploymentLocalGitUrl": "https://<username>@<app_name>.scm.azurewebsites.net/<app_name>.git",
  "enabled": true,
  < JSON data removed for brevity. >
}
```

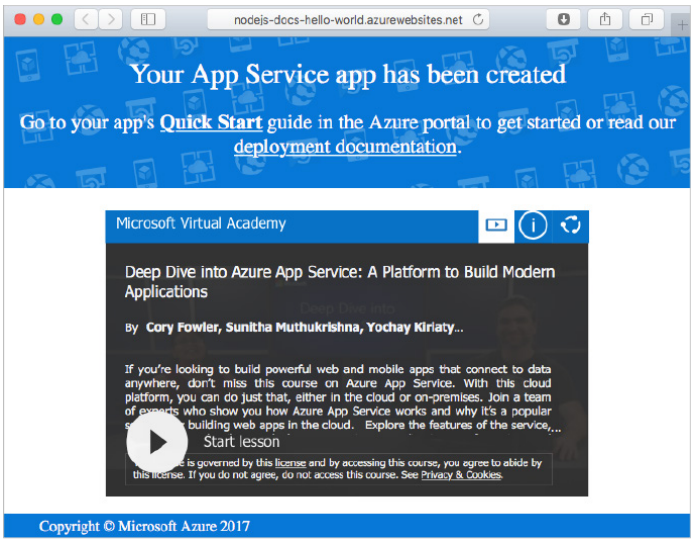
You've created an empty web app, with git deployment enabled.

NOTE

The URL of the Git remote is shown in the deploymentLocalGitUrl property, with the format https://<username>@<app_name>.scm.azurewebsites.net/<app_name>.git. Save this URL as you'll need it later.

Browse to your newly created web app. Replace `<app name>` with a unique app name.

```
http://<app name>.azurewebsites.net
```



Push to Azure from Git

In the local terminal window, add an Azure remote to your local Git repository. This Azure remote was created for you in Create a web app.

```
git remote add azure <deploymentLocalGitUrl-from-create-step>
```

Push to the Azure remote to deploy your app with the following command. When prompted for a password, make sure that you enter the password you created in Configure a deployment user, not the password you use to log in to the Azure portal.

```
git push azure master
```

The preceding command displays information similar to the following example:

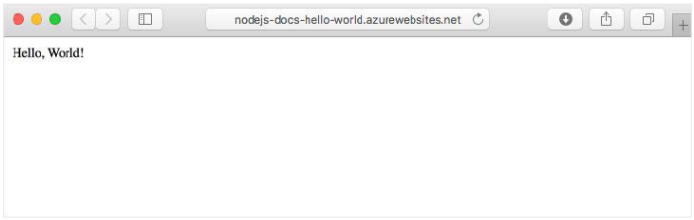
```
Counting objects: 23, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (23/23), 3.71 KiB | 0 bytes/s, done.
Total 23 (delta 8), reused 7 (delta 1)
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id 'bf114df591'.
remote: Generating deployment script.
remote: Generating deployment script for node.js Web Site
remote: Generated deployment script files
remote: Running deployment command...
remote: Handling node.js deployment.
remote: Kudu sync from: '/home/site/repository' to: '/home/site/wwwroot'
remote: Copying file: '.gitignore'
remote: Copying file: 'LICENSE'
remote: Copying file: 'README.md'
remote: Copying file: 'index.js'
remote: Copying file: 'package.json'
remote: Copying file: 'process.json'
remote: Deleting file: 'hostingstart.html'
remote: Ignoring: .git
remote: Using start-up script index.js from package.json.
remote: Node.js versions available on the platform are: 4.4.7, 4.5.0, 6.2.2, 6.6.0, 6.9.1.
remote: Selected node.js version 6.9.1. Use package.json file to choose a different version.
remote: Selected npm version 3.10.8
remote: Finished successfully.
remote: Running post deployment command(s)...
remote: Deployment successful.
To https://<app_name>.scm.azurewebsites.net:443/<app_name>.git
 * [new branch]      master -> master
```

Browse to the app

Browse to the deployed application using your web browser.

```
http://<app_name>.azurewebsites.net
```

The Node.js sample code is running in a web app with built-in image.



Congratulations! You've deployed your first Node.js app to App Service on Linux.

Update and manage

Now that you've deployed your app, here's how to make updates and manage it.

Update and redeploy the code

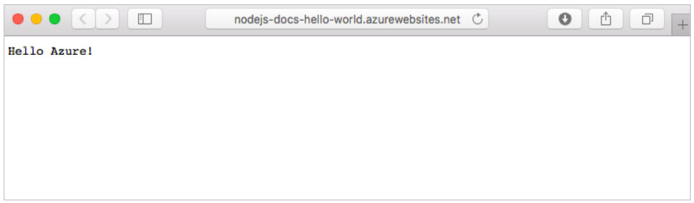
Using a text editor, open the index.js file in the Node.js app, and make a small change to the text in the call to response.en .

```
response.end("Hello Azure!");
```

Commit your changes in Git, and then push the code changes to Azure.

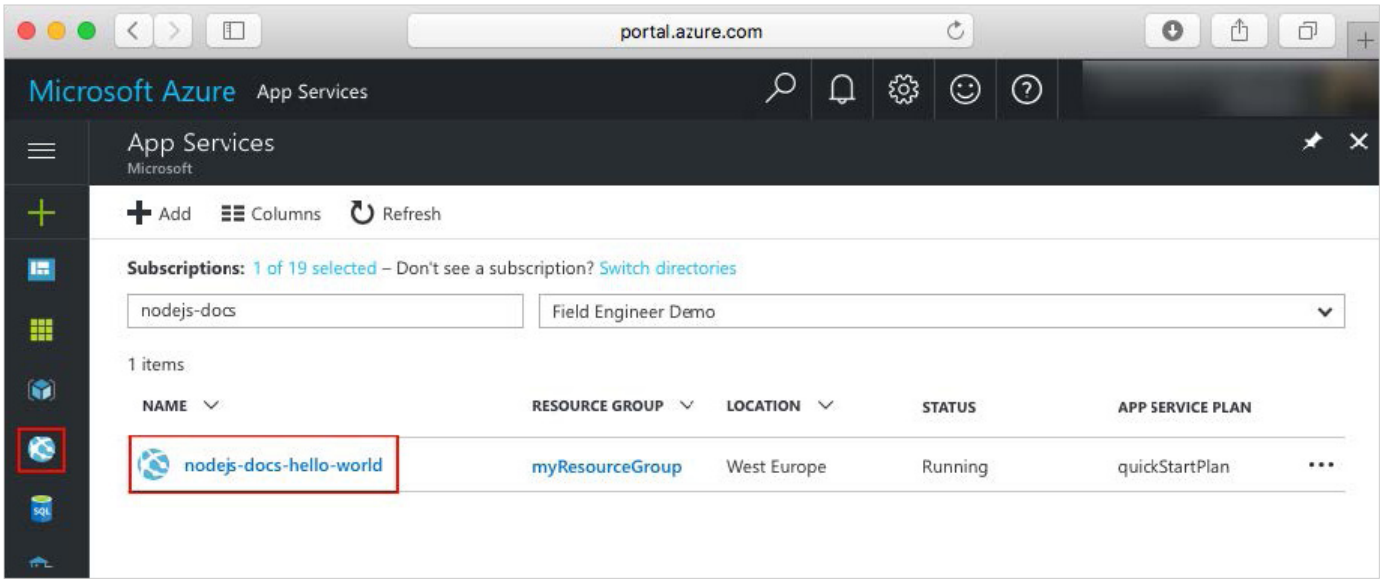
```
git commit -am "updated output"
git push azure master
```

Once deployment has completed, switch back to the browser window that opened in the Browse to the app step, and hit refresh.

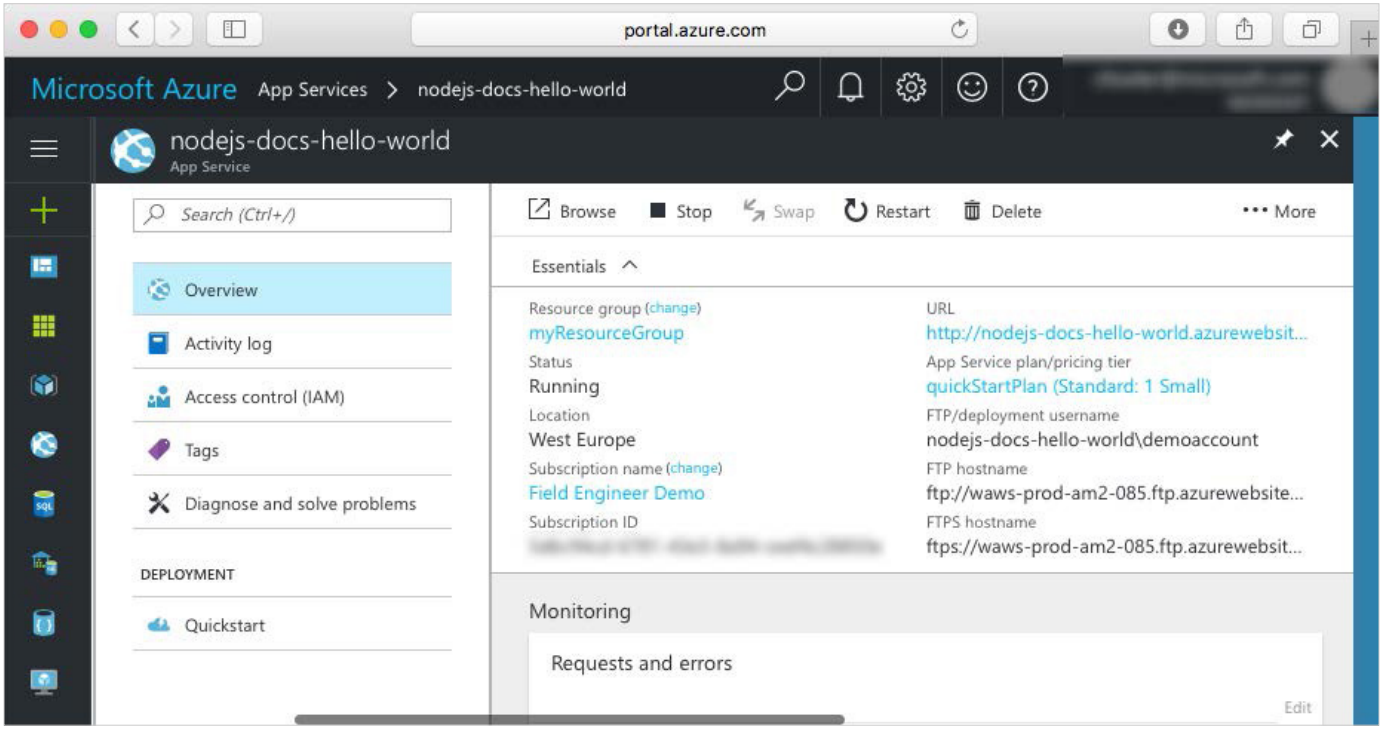


Manage your new Azure web app

Go to the Azure portal to manage the web app you created. From the left menu, click App Services, and then click the name of your Azure web app.



You see your web app's Overview page. Here, you can perform basic management tasks like browse, stop, start, restart, and delete.



In the left menu, there are different pages for configuring your app.

Clean up resources

To clean up your resources, run the following command:

```
az group delete --name myResourceGroup
```

Now that you have created the front end, we can look at the back end of your application.

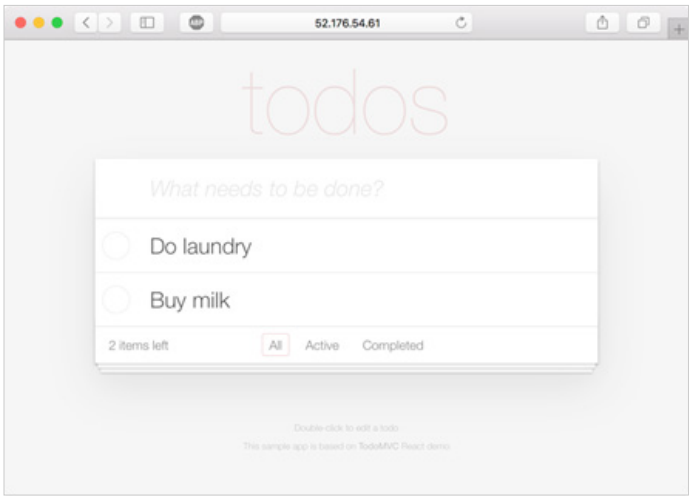
Deploy an Azure Container Service (AKS) cluster

Now that you are familiar with using Node.js with Azure, you can now work with more concepts and functionality that will enhance your web apps.

Azure Container Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise.

A Docker container is a lightweight, stand-alone, executable, logical package of software. It includes everything needed to run the software: code, runtime, system tools, system libraries, and settings so that the software within is isolated from its surroundings. Using containers eliminate the problems created when there are differences between development and staging environment. This helps with reducing conflicts and allows for more predictable deployment of new features and functionality to the production environment. For more information about containers, see the Docker info page. Containers eliminate the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline.

In this example, you will deploy an AKS cluster using the Azure CLI. You will then run a simple Todo application written in Node.js, which stores data on the client using HTML5 local storage. Once completed, you can access the application over the internet.



This example assumes a basic understanding of Kubernetes concepts, for detailed information on Kubernetes see the Kubernetes documentation.

Get set up

Launch Azure Cloud Shell

Click the Cloud Shell button on the menu in the upper-right of the Azure portal.

Enabling AKS preview for your Azure subscription (temporary)

As of writing, AKS is still in a preview state. While AKS is in preview, you need a feature flag on your subscription for creating new clusters. You can request this feature for any number of subscriptions that you would like to use. Use the `az provider register` command to register the AKS provider:

```
az provider register -n Microsoft.ContainerService
```

After registering, you are now ready to create a Kubernetes cluster with AKS.

Create a resource group

Create a resource group with the az group create command.

Use the following example to create a resource group named *myResourceGroup* in the *Central US* location.

```
{
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myResourceGroup",
  "location": "centralus",
  "managedBy": null,
  "name": "myResourceGroup",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

Create AKS cluster

Use the following example to create a cluster named *myK8sCluster* with one node.

```
az aks create \
  --resource-group myResourceGroup \
  --name myK8sCluster \
  --node-count 1 \
  --generate-ssh-keys
```

After several minutes, the command completes and returns JSON-formatted information about the cluster.

Deploy and test

Connect to the cluster

To manage a Kubernetes cluster, use kubectl, the Kubernetes command-line client.

If you're using Azure Cloud Shell, kubectl is already installed. If you want to install it locally, run the following command.

```
az aks install-cli
```

Run the following command to configure kubectl to connect to your Kubernetes cluster. This step downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myResourceGroup --name myK8sCluster
```

To verify the connection to your cluster, use the kubectl get command to return a list of the cluster nodes.

```
  kubectl get nodes
```

Output:

```
NAME                          STATUS    ROLES     AGE       VERSION
k8s-myk8scluster-36346190-0   Ready     agent     2m        v1.7.7
```

Run the application

A Kubernetes manifest file defines a desired state for the cluster, including what container images should be running. For this example, a manifest is used to create all objects needed to run the Todo application.

Create a Kubernetes manifest file, called `todo-app.yaml`, by copying in the following YAML code. If you are working in Azure Cloud Shell, you can create this file using vi or nano as if working on a virtual or physical system.

```
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    name: web
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 3000
      protocol: TCP
  selector:
    name: web
---
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: web
  name: web-controller
spec:
  replicas: 2
  selector:
    name: web
  template:
    metadata:
      labels:
        name: web
    spec:
      containers:
      - image: nodeebookdemo/nodejs-todo-sample
        name: web
        ports:
        - containerPort: 3000
          name: http-server
```

Use the kubectl create command to run the application.

```
kubectl create -f todo-app.yaml
```

Output:

```
service "web" created
replicationcontroller "web-controller" created
```

Test the application

As the application is run, a Kubernetes service is created that exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the kubectl get service command with the `--watch` argument.

```
kubectl get service web --watch
```

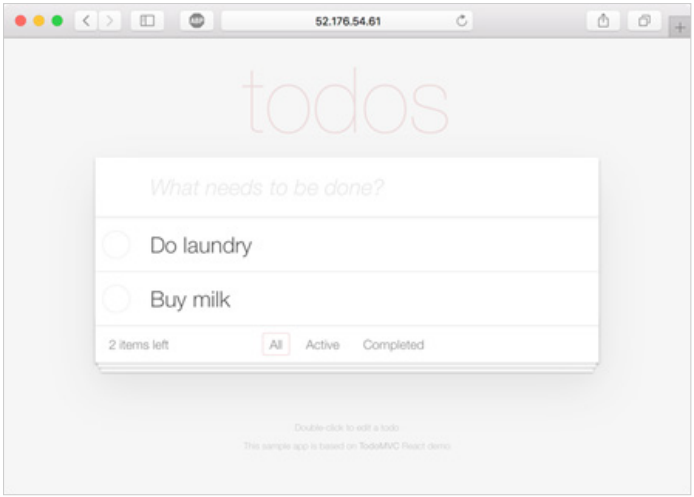
Initially the EXTERNAL-IP for the azure-vote-front service appears as pending.

```
NAME              TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
web               LoadBalancer   10.0.37.27   <pending>     80:30572/TCP   6s
```

Once the EXTERNAL-IP address has changed from pending to an IP address, use `Ctrl+C` to stop the kubectl watch process.

```
NAME              TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)        AGE
web               LoadBalancer   10.0.37.27       52.179.23.131    80:30572/TCP   2m
```

You can now browse to the external IP address to see the Todo App.



Finish up

Delete cluster

When you no longer need the cluster, you can use the az group delete command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup --yes --no-wait
```

Get the code

In this example, you used pre-created container images to create a Kubernetes deployment. You can get the related application code, Dockerfile, and Kubernetes manifest file on GitHub.

https://github.com/NodeEbookDemo/nodejs-todo-sample

Next steps

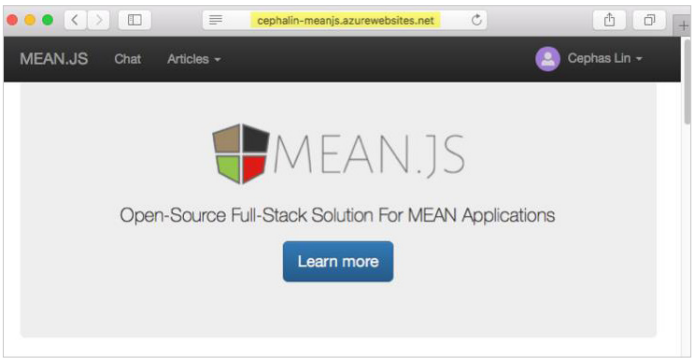
Now that you know about applications let's talk about the data.

Azure Cosmos DB: Migrate an existing Node.js Mongo DB web app

Azure Cosmos DB is Microsoft's globally distributed multi-model database service, and is compatible with MongoDB. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This example demonstrates how to use an existing app written in Node.js and designed to use MongoDB, and connect it to your Azure Cosmos DB database, which supports MongoDB client connections. In other words, your Node.js application only knows that it's connecting to a database using MongoDB APIs. It is transparent to the application that the data is stored in Azure Cosmos DB.

When you are done, you will have a MEAN application (MongoDB, Express, Angular, and Node.js) running on Azure Cosmos DB.



Before you begin

Get the software you need

Install the software and get an Azure account as described here.

Launch Azure Cloud Shell

Click the **Cloud Shell** button on the menu in the upper-right of the Azure portal.

Get set up

Clone the sample application

Open a git terminal window, such as git bash, and `cd` to a working directory.

Run the following commands to clone the sample repository. This sample repository contains the default MEAN.js application.

```
git clone https://github.com/prashanthmadi/mean
```

Run the application

Install the required packages and start the application.

```
cd mean
npm install
npm start
```

The application will try to connect to a MongoDB source and fail. Exit the application when the output returns "[MongoError: connect ECONNREFUSED 127.0.0.1:27017]".

Create a resource group

Create a resource group with the az group create.

The following example creates a resource group in the West Europe region. Choose a unique name for the resource group.

```
az group create --name myResourceGroup --location "West Europe"
```

Create an Azure Cosmos DB account

Create an Azure Cosmos DB account with the *az cosmosdb* create command.

In the following command, substitute your own unique Azure Cosmos DB account name where you see the `<cosmosdb-name>` placeholder. This unique name will be used as part of your Azure Cosmos DB endpoint (`https://<cosmosdb-name>.documents.azure.com/`), so the name needs to be unique across all Azure Cosmos DB accounts in Azure.

```
az cosmosdb create --name <cosmosdb-name> --resource-group myResourceGroup --kind MongoDB
```

Use the `--kind MongoDB` parameter to enable MongoDB client connections.

When you have created the Azure Cosmos DB, the Azure CLI shows information similar to the following example.

```
{
  "databaseAccountOfferType": "Standard",
  "documentEndpoint": "https://<cosmosdb-name>.documents.azure.com:443/",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myResourceGroup/providers/Microsoft.DocumentDB/databaseAccounts/<cosmosdb-name>",
  "kind": "MongoDB",
  "location": "West Europe",
  "name": "<cosmosdb-name>",
  "readLocations": [
    {
      "documentEndpoint": "https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ],
  "resourceGroup": "myResourceGroup",
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "writeLocations": [
    {
      "documentEndpoint": "https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ]
}
```

Connect and migrate

In this step, you connect your MEAN.js sample application to an Azure Cosmos DB database you just created, using a MongoDB connection string.

Configure the connection string in your Node.js application

Open `config/env/local-development.js` In your MEAN.js repository.

Replace the content of this file with the following code. Be sure to also replace the two `<cosmosdb-name>` placeholders with your Azure Cosmos DB account name.

```
'use strict';
module.exports = {
  db: {
    uri: 'mongodb://<cosmosdb-name>:<primary_master_key>@<cosmosdb-name>.documents.azure.com:10255/
mean-dev?ssl=true&sslverifycertificate=false'
  }
};
```

Retrieve the key

In order to connect to an Azure Cosmos DB database, you need the database key. Use the [az cosmosdb list-keys] command to retrieve the primary key.

```
az cosmosdb list-keys \
  --name <cosmosdb-name> \
  --resource-group myResourceGroup \
  --query "primaryMasterKey"
```

The Azure CLI outputs information similar to the following example.

```
"RUayjYjixJDWG5xTqIiXjC..."
```

Copy the value of `primaryMasterKey`. Paste this over the `<primary_master_key>` in `local-development.js`.

Save your changes.

Run the application again

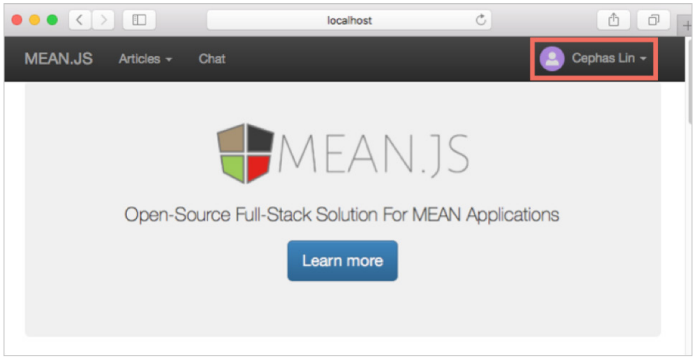
Run `npm start` again.

```
npm start
```

A console message should indicate that the development environment is up and running.

Navigate to `http://localhost:3000` in a browser. Click Sign Up in the top menu and try to create two dummy users.

The MEAN.js sample application stores user data in the database. If MEAN.js automatically signs into the created user, then your Azure Cosmos DB connection is working.



View data in Data Explorer

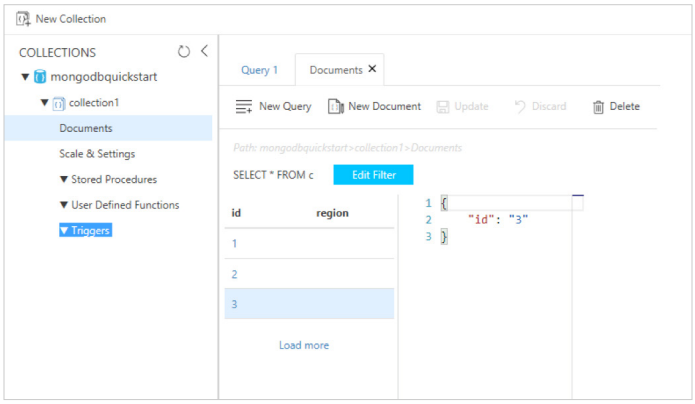
You can view, query, and run business-logic on data stored by an Azure Cosmos DB in the Azure portal. To view, query, and work with the user data created in the previous step, login to the Azure portal in your web browser.

1. In the top Search box, type Azure Cosmos DB.

2. When your Cosmos DB account blade opens, select your Cosmos DB account.

3. In the left navigation, click Data Explorer,

4. Expand your collection in the Collections pane, and then you can view the documents in the collection, query the data, and even create and run stored procedures, triggers, and UDFs.



Deploy the Node.js application to Azure

In this step, you will deploy your MongoDB-connected Node.js application to Azure Cosmos DB.

You may have noticed that the configuration file that you changed earlier is for the development environment (`/config/env/local-development.js`). When you deploy your application to App Service, it will run in the production environment by default. So now, you need to make the same change to the respective configuration file.

In your MEAN.js repository, open `config/env/production.js`.

In the `db object`, replace the value of `uri` as show in the following example. Be sure to replace the placeholders as before.

```
'mongodb://<cosmosdb-name>:<primary_master_key>@<cosmosdb-name>.documents.azure.com:10255/
mean?ssl=true&sslverifycertificate=false',
```

NOTE

The `ssl=true` option is important because Azure Cosmos DB requires SSL.

In the terminal, commit all your changes into Git.

```
git add .
git commit -m "configured MongoDB connection string"
```

Clean up resources

If you're not going to continue to use this app, delete all resources created by this example in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and click the name of the resource you created.

2. On your resource group page, click **Delete**. Type the name of the resource to delete in the text box, and click **Delete**.

Next steps

In this example, you've learned how to create an Azure Cosmos DB account and create a MongoDB collection using the Data Explorer. You can now migrate your MongoDB data to Azure Cosmos DB.

If your application uses a relational database you can look at using MySQL. The same instructions discussed below will also be applicable to PostgreSQL with some minor changes.

Azure Database for MySQL: Use Node.js to connect and query data

This example demonstrates how to connect to an Azure Database for MySQL using Node.js. It shows how to use SQL statements to query, insert, update, and delete data in the database.

Get set up

This quickstart uses the resources created in either of these guides as a starting point:

- Create an Azure Database for MySQL server using Azure portal
- Create an Azure Database for MySQL server using Azure CLI

Add the mysql2 NPM package

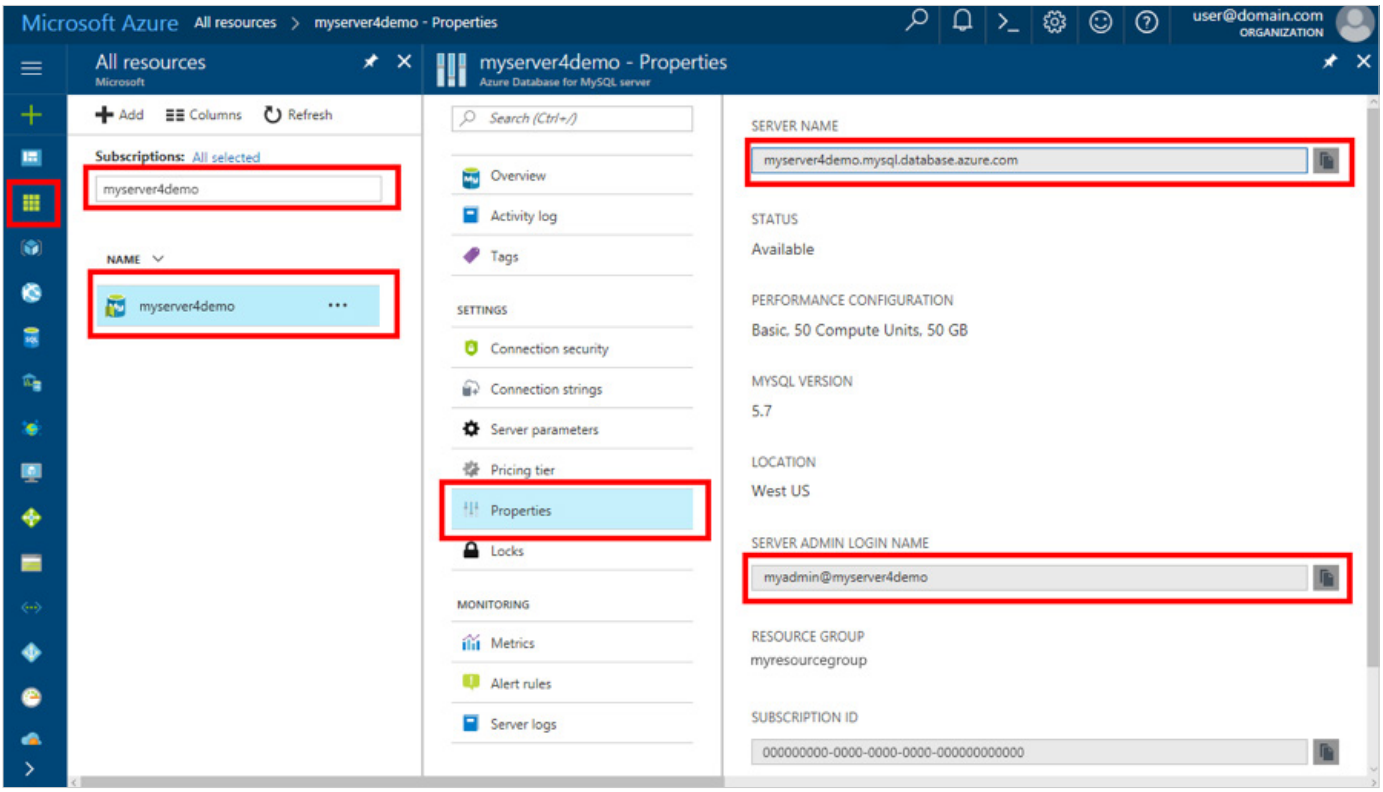
In the folder containing an existing Node.js app, add the *mysql2* NPM package:

```
npm install --save mysql2
```

Prepare to connect

Get connection information
Get the connection information needed to connect to the Azure Database for MySQL. You need the fully qualified server name and login credentials.

1. Log in to the Azure portal.
2. In the left pane, click **All resources**, and then search for the server you have created (for example, **myserver4demo**).
3. Click the server name **myserver4demo**.
4. Select the server's **Properties** page, and then make a note of the **Server name** and **Server admin login name.**
5. If you forget your server login information, navigate to the Overview page to view the Server admin login name, and reset the password.



Running the JavaScript code in Node.js

1. Paste the JavaScript code into text files, and then save it into a project folder with file extension .js such as `C:\nodejsmysql\createtable.js` or `/home/username/nodejsmysql/createtable.js`.

2. Launch the command prompt or bash shell, and then change directory into your project folder `cd nodejsmysql`.

3. To run the application, type the node command followed by the file name, such as node `createtable.js`.

4. On Windows, if the node application is not in your environment variable path, you may need to use the full path to launch the node application, such as node createtable.js.

Connect and manipulate data

Connect, create table, and insert data

Use the following code to connect and load the data by using **CREATE TABLE** and **INSERT INTO** SQL statements.

- Use the mysql.createConnection() method to interface with the MySQL server
- Use the connect() function to establish the connection to the server
- Use the query() function to execute the SQL query against MySQL database

Replace the `host`, `user`, `password`, and `database` parameters with the values that you specified when you created the server and database.

```
const mysql = require('mysql2');

var config =
{
    host: 'myserver4demo.mysql.database.azure.com',
    user: 'myadmin@myserver4demo',
    password: 'your_password',
    database: 'quickstartdb',
    port: 3306,
    ssl: true
};

const conn = new mysql.createConnection(config);

conn.connect(
    function (err) {
    if (err) {
        console.log("!!! Cannot connect !!! Error:");
        throw err;
    }
    else
    {
       console.log("Connection established.");
           queryDatabase();
    }
});

function queryDatabase(){
       conn.query('DROP TABLE IF EXISTS inventory;', function (err, results, fields) {
            if (err) throw err;
            console.log('Dropped inventory table if existed.');
        })
       conn.query('CREATE TABLE inventory (id serial PRIMARY KEY, name VARCHAR(50), quantity INTEGER);',
            function (err, results, fields) {
                if (err) throw err;
            console.log('Created inventory table.');
        })
       conn.query('INSERT INTO inventory (name, quantity) VALUES (?, ?);', ['banana', 150],
            function (err, results, fields) {
                if (err) throw err;
            else console.log('Inserted ' + results.affectedRows + ' row(s).');
        })
       conn.query('INSERT INTO inventory (name, quantity) VALUES (?, ?);', ['orange', 154],
            function (err, results, fields) {
                if (err) throw err;
            console.log('Inserted ' + results.affectedRows + ' row(s).');
        })
       conn.query('INSERT INTO inventory (name, quantity) VALUES (?, ?);', ['apple', 100],
        function (err, results, fields) {
                if (err) throw err;
            console.log('Inserted ' + results.affectedRows + ' row(s).');
        })
       conn.end(function (err) {
        if (err) throw err;
        else  console.log('Done.')
        });
};
```

Read data

Use the following code to connect and read the data by using a **SELECT** SQL statement.

- Use the mysql.createConnection() method to interface with the MySQL server
- Use the connect() function to establish the connection to the server
- Use the query() function to execute the SQL query against MySQL database
- Use the results array to hold the results of the query

Replace the `host`, `user`, `password`, and `database` parameters with the values that you specified when you created the server and database.

```
const mysql = require('mysql2');

var config =
{
    host: 'myserver4demo.mysql.database.azure.com',
    user: 'myadmin@myserver4demo',
    password: 'your_password',
    database: 'quickstartdb',
    port: 3306,
    ssl: true
};

const conn = new mysql.createConnection(config);

conn.connect(
    function (err) {
        if (err) {
            console.log("!!! Cannot connect !!! Error:");
            throw err;
        }
        else {
            console.log("Connection established.");
            readData();
        }
    });

function readData(){
        conn.query('SELECT * FROM inventory',
            function (err, results, fields) {
                if (err) throw err;
                else console.log('Selected ' + results.length + ' row(s).');
                for (i = 0; i < results.length; i++) {
                    console.log('Row: ' + JSON.stringify(results[i]));
                }
                console.log('Done.');
            })
       conn.end(
           function (err) {
                if (err) throw err;
                else  console.log('Closing connection.')
        });
};
```

Update data

Use the following code to connect and read the data by using an **UPDATE** SQL statement.

- Use the mysql.createConnection() method to interface with the MySQL server
- Use the connect() function to establish the connection to the server
- Use the query() function to execute the SQL query against MySQL database
- Use the results array to hold the results of the query

Replace the `host`, `user`, `password`, and `database` parameters with the values that you specified when you created the server and database.

```
const mysql = require('mysql2');

var config =
{
    host: 'myserver4demo.mysql.database.azure.com',
    user: 'myadmin@myserver4demo',
    password: 'your_password',
    database: 'quickstartdb',
    port: 3306,
    ssl: true
};

const conn = new mysql.createConnection(config);

conn.connect(
    function (err) {
        if (err) {
            console.log("!!! Cannot connect !!! Error:");
            throw err;
        }
        else {
            console.log("Connection established.");
            updateData();
        }
    });

function updateData(){
       conn.query('UPDATE inventory SET quantity = ? WHERE name = ?', [200, 'banana'],
            function (err, results, fields) {
                if (err) throw err;
                else console.log('Updated ' + results.affectedRows + ' row(s).');
        })
       conn.end(
           function (err) {
                if (err) throw err;
                else  console.log('Done.')
        });
};
```

Delete data

Use the following code to connect and read the data by using an **DELETE** SQL statement.

- Use the mysql.createConnection() method to interface with the MySQL server
- Use the connect() function to establish the connection to the server
- Use the query() function to execute the SQL query against MySQL database
- Use the results array to hold the results of the query

Replace the `host`, `user`, `password`, and `database` parameters with the values that you specified when you created the server and database.

```
const mysql = require('mysql2');

var config =
{
    host: 'myserver4demo.mysql.database.azure.com',
    user: 'myadmin@myserver4demo',
    password: 'your_password',
    database: 'quickstartdb',
    port: 3306,
    ssl: true
};

const conn = new mysql.createConnection(config);

conn.connect(
    function (err) {
        if (err) {
            console.log("!!! Cannot connect !!! Error:");
            throw err;
        }
        else {
            console.log("Connection established.");
            deleteData();
        }
    });

function deleteData(){
       conn.query('DELETE FROM inventory WHERE name = ?', ['orange'],
            function (err, results, fields) {
                if (err) throw err;
                else console.log('Deleted ' + results.affectedRows + ' row(s).');
        })
       conn.end(
           function (err) {
                if (err) throw err;
                else  console.log('Done.')
        });
};
```

Next steps

Now that we've implemented databases in our applications, we can include Redis Cache to improve the performance of our services. Redis Cache is based on the open source Redis project. It's a fast, in-memory, key-value store that can be used to cache data from the database or from other sources.

How to use Azure Redis Cache with Node.js

Azure Redis Cache gives you access to a secure, dedicated Redis cache, managed by Microsoft. Redis is an open source, advanced key-value store, that can contain strings, hashes, lists, sets and sorted sets. Your cache is accessible from any application within Microsoft Azure.

This topic shows you how to get started with Azure Redis Cache using Node.js.

Get set up

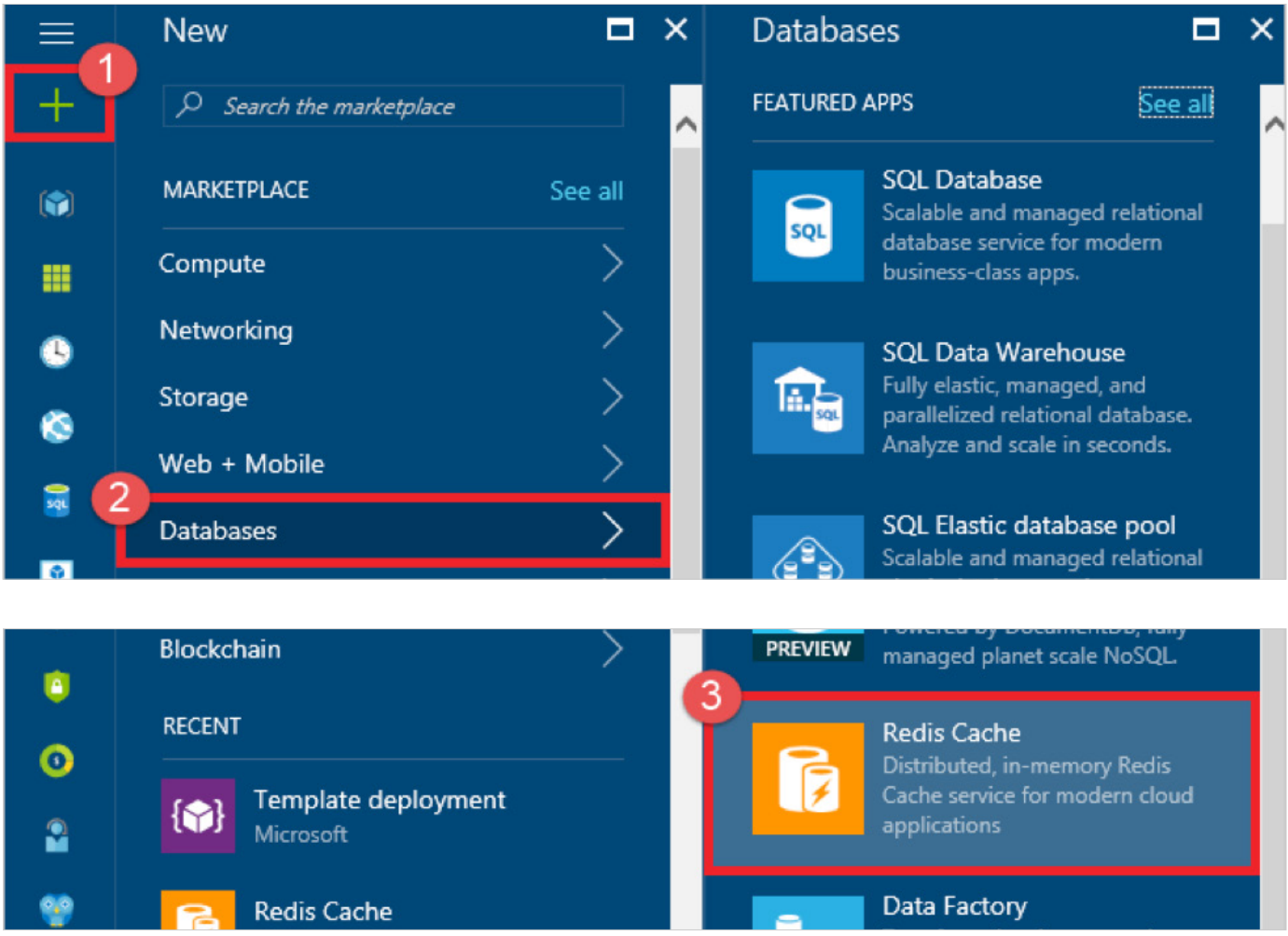
Install node_redis:

```
npm install redis
```

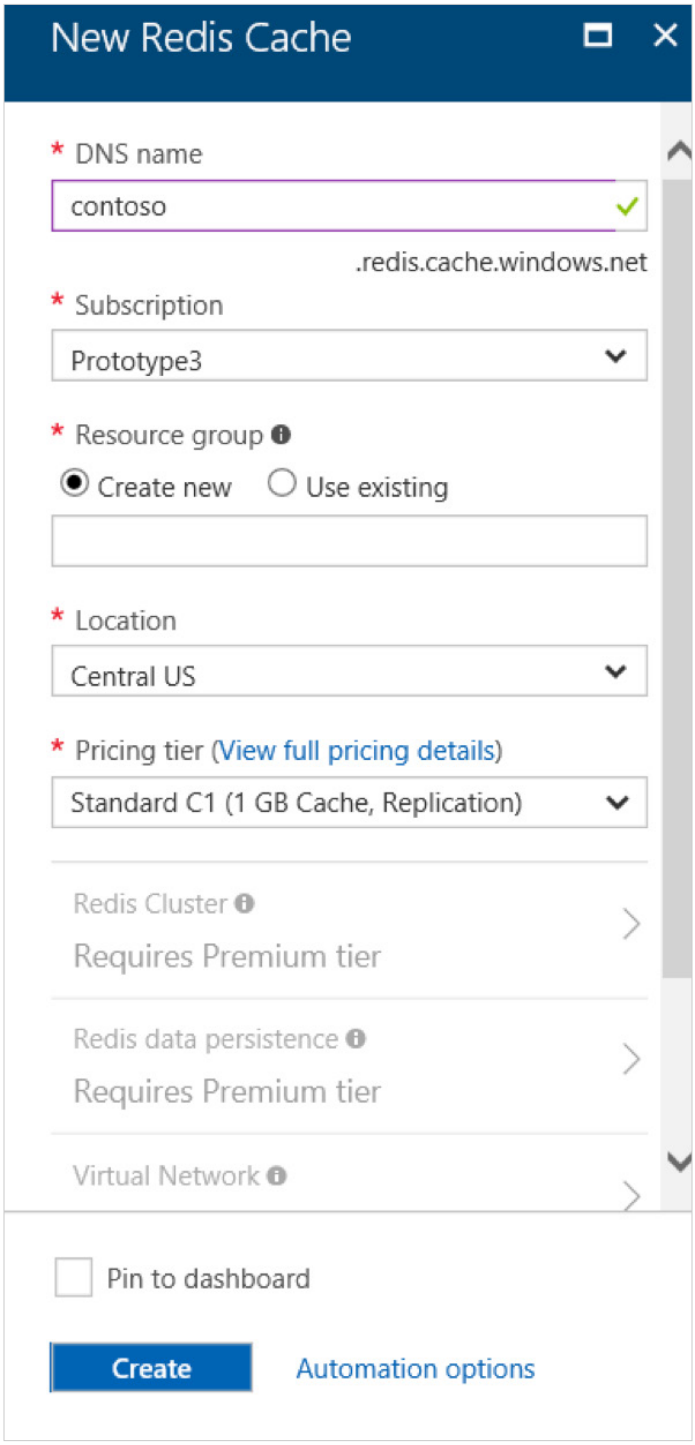
This example uses node_redis. For examples of using other Node.js clients, see the individual documentation for the Node.js clients listed at Node.js Redis clients.

Create a Redis cache on Azure

To create a cache, first sign in to the Azure portal, and click **New > Databases > Redis Cache**.



In the **New Redis Cache** blade, specify the desired configuration for the cache.



- In **DNS name**, enter a unique cache name to use for the cache endpoint. The cache name must be a string between 1 and 63 characters and contain only numbers, letters, and the `-` character. The cache name cannot start or end with the `-` character, and consecutive `-` characters are not valid.

- For **Subscription**, select the Azure subscription that you want to use for the cache. If your account has only one subscription, it will be automatically selected and the **Subscription** drop-down will not be displayed.

- In **Resource group**, select or create a resource group for your cache.

- Use **Location** to specify the geographic location in which your cache is hosted. For the best performance, Microsoft strongly recommends that you create the cache in the same region where yourapplication is deployed.

- Use **Pricing tier** to select the desired cache size and features.

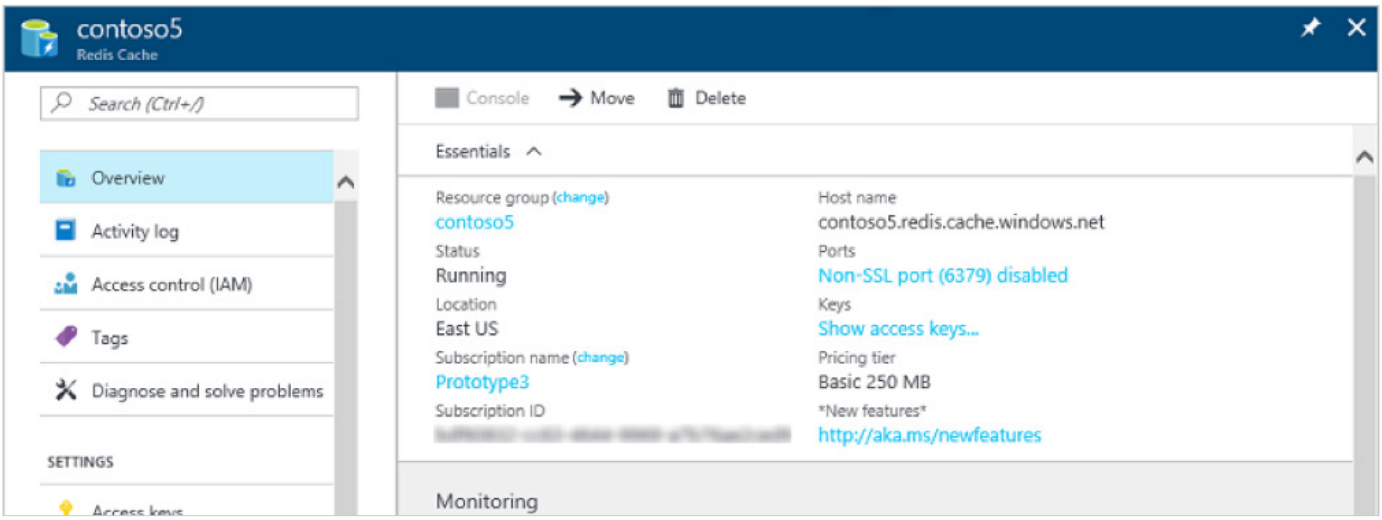
- **Redis cluster** allows you to create caches larger than 53 GB and to shard data across multiple Redis nodes. For more information, see How to configure clustering for a Premium Azure Redis Cache.

- **Redis persistence** offers the ability to persist your cache to an Azure Storage account. For instructions on configuring persistence, see How to configure persistence for a Premium Azure Redis Cache.

- **Virtual Network** provides enhanced security and isolation by restricting access to your cache to only those clients within the specified Azure Virtual Network. You can use all the features of VNet such as subnets, access control policies, and other features to further restrict access to Redis. For more information, see How to configure Virtual Network support for a Premium Azure Redis Cache.

- By default, non-SSL access is disabled for new caches. To enable the non-SSL port, check **Unblock port 6379 (not SSL encrypted)**; this is optional.

Once the new cache options are configured, click Create. It can take a few minutes for the cache to be created. To check the status, you can monitor the progress on the startboard. After the cache has been created, your new cache has a **Running** status and is ready for use with default settings.

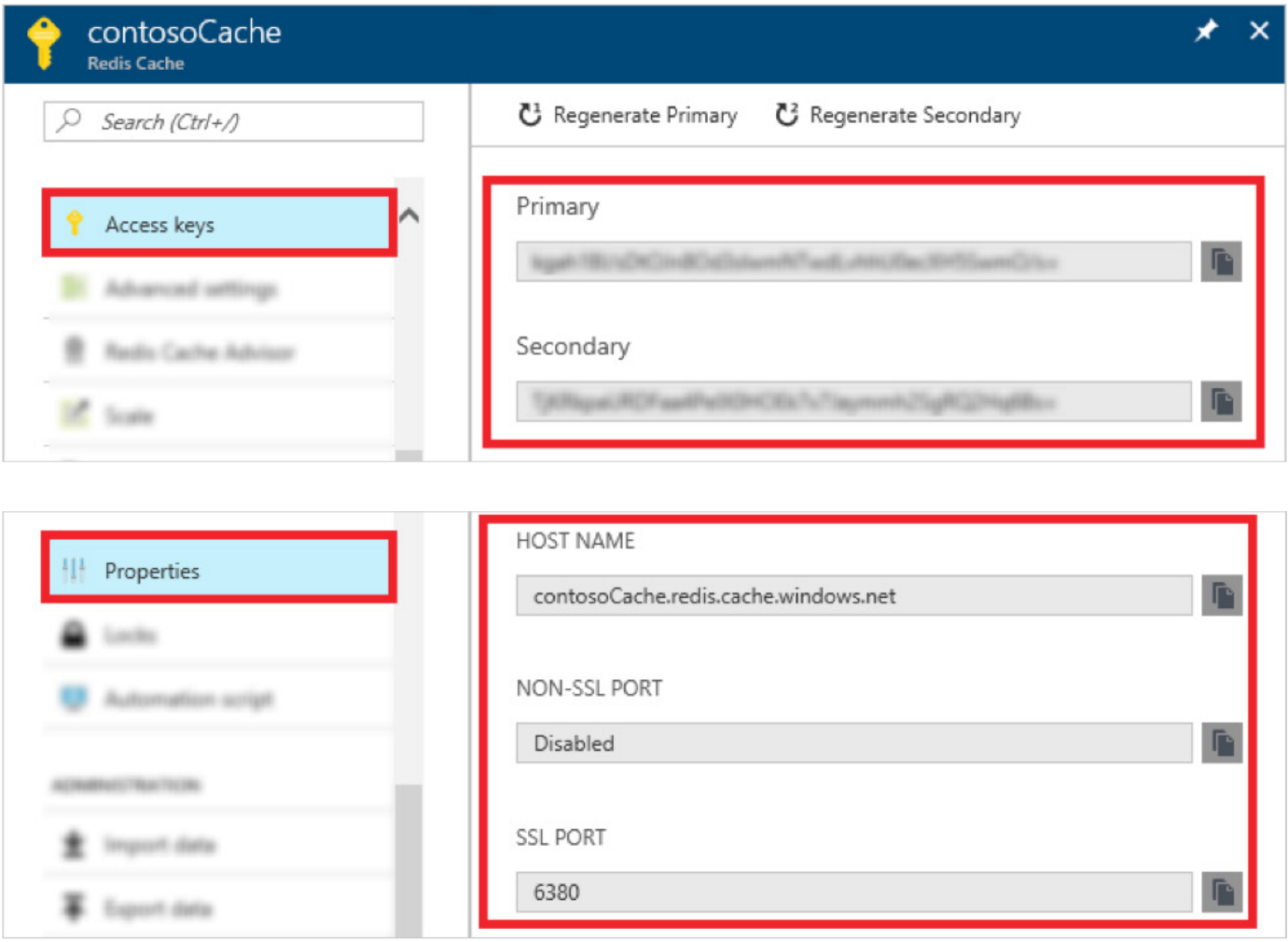


Retrieve the host name and access keys

To connect to an Azure Redis Cache instance, cache clients need the host name, ports, and keys of the cache. Some clients may refer to these items by slightly different names. You can retrieve this information in the Azure portal or by using command-line tools such as Azure CLI.

Retrieve host name, ports, and access keys using the Azure Portal

To retrieve host name, ports, and access keys using the Azure Portal, browse to your cache in the Azure portal and click **Access keys** and **Properties** in the **Resource menu.**



Retrieve host name, ports, and access keys using Azure CLI

To retrieve the host name and ports using Azure CLI 2.0 you can call az redis show, and to retrieve the keys you can call az redis list-keys. The following script calls these two commands and prints the hostname, ports, and keys to the console.

```
#!/bin/bash

# Retrieve the hostname, ports, and keys for contosoCache located in contosoGroup

# Retrieve the hostname and ports for an Azure Redis Cache instance
redis=($(az redis show --name contosoCache --resource-group contosoGroup --query [hostName,enableNonSslPort,port,sslPort] --output tsv))

# Retrieve the keys for an Azure Redis Cache instance
keys=($(az redis list-keys --name contosoCache --resource-group contosoGroup --query [primaryKey,secondaryKey] --output tsv))

# Display the retrieved hostname, keys, and ports
echo "Hostname:" ${redis[0]}
echo "Non SSL Port:" ${redis[2]}
echo "Non SSL Port Enabled:" ${redis[1]}
echo "SSL Port:" ${redis[3]}
echo "Primary Key:" ${keys[0]}
echo "Secondary Key:" ${keys[1]}
```

For more information about this script, see Get the hostname, ports, and keys for Azure Redis Cache.

Connect to the cache securely using SSL

The latest builds of node_redis provide support for connecting to Azure Redis Cache using SSL. The following example shows how to connect to Azure Redis Cache using the SSL endpoint of 6380. Replace `<name>` with the name of your cache and `<key>` with either your primary or secondary key as described in the previous Retrieve the host name and access keys section.

```
var redis = require("redis");

  // Add your cache name and access key.
var client = redis.createClient(6380,'<name>.redis.cache.windows.net', {auth_pass: '<key>', tls:
{servername: '<name>.redis.cache.windows.net'}});
```

NOTE

The non-SSL port is disabled for new Azure Redis Cache instances. If you are using a different client that doesn't support SSL, see How to enable the non-SSL port.

Add something to the cache and retrieve it

The following example shows you how to connect to an Azure Redis Cache instance, and store and retrieve an item from the cache. For more examples of using Redis with the node_redis client, see http://redis.js.org/.

```
var redis = require("redis");

  // Add your cache name and access key.
var client = redis.createClient(6380,'<name>.redis.cache.windows.net', {auth_pass: '<key>', tls:
{servername: '<name>.redis.cache.windows.net'}});

client.set("key1", "value", function(err, reply) {
        console.log(reply);
    });

client.get("key1",  function(err, reply) {
        console.log(reply);
    });
```

Output:

```
OK
value
```

Next steps

- Enable cache diagnostics so you can monitor the health of your cache
- Read the official Redis documentation

Finally, let's explore how we can use Azure Blob Storage. Many applications need an object storage where they can store any kind of unstructured data, including documents, photos, and attachments. Using the Azure Storage SDK for Node.js developers can use Azure Blob Storage to store these files quickly and in a cost-effective way, using the Azure Storage SDK for Node.js.

How to use Blob storage from Node.js

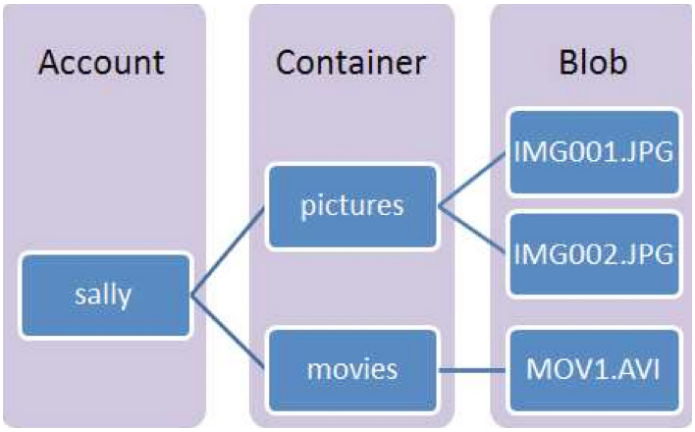
In this section, you will perform common scenarios using Blob storage. The samples are written via the Node.js API. The scenarios covered include how to upload, list, download, and delete blobs.

Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- Serving images or documents directly to a browser
- Storing files for distributed access
- Streaming video and audio
- Storing data for backup and restore, disaster recovery, and archiving
- Storing data for analysis by an on-premises or Azure-hosted service

The Blob service contains the following components:



- **Storage Account:** Use a storage account for All access to Azure Storage. You can use a **General-purpose storage account** or a **Blob storage account**, which is specialized for storing objects/ blobs. See About Azure storage accounts for more information.

- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. (The container name must be lowercase.)

- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs. For this exercise we are concerned only with block and append blobs.

    - *Block blobs* are ideal for storing text or binary files, such as documents and media files. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000).

    - *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

    - *Page blobs* can be up to 8 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see Naming and Referencing Containers, Blobs, and Metadata.

Configure your application to access storage

To use Azure storage, you need the Azure Storage SDK for Node.js, which includes a set of convenience libraries that communicate with the storage REST services.

Use Node Package Manager (NPM) to obtain the package

1. Use a command-line interface to navigate to the folder where you created your sample application.

2. Type **npm install --save azure-storage** in the command window. Output from the command is similar to the following code example.

```
azure-storage@0.5.0 node_modules\azure-storage
 +-- extend@1.2.1
 +-- xmlbuilder@0.4.3
 +-- mime@1.2.11
 +-- node-uuid@1.4.3
 +-- validator@3.22.2
 +-- underscore@1.4.4
 +-- readable-stream@1.0.33 (string_decoder@0.10.31, isarray@0.0.1, inherits@2.0.1, core-util-is@1.0.1)
 +-- xml2js@0.2.7 (sax@0.5.2)
 +-- request@2.57.0 (caseless@0.10.0, aws-sign2@0.5.0, forever-agent@0.6.1, stringstream@0.0.4, oauth-sign@0.8.0, tun-
nel-agent@0.4.1, isstream@0.1.2, json-stringify-safe@5.0.1, bl@0.9.4, combined-stream@1.0.5, qs@3.1.0, mime-types@2.0.14,
form-data@0.2.0, http-signature@0.11.0, tough-cookie@2.0.0, hawk@2.3.1, har-validator@1.8.0)
```

Import the package

Using Notepad or another text editor, add the following to the top of the file of the application where you intend to use storage:

```
var azure = require('azure-storage');
```

Set up an Azure Storage connection

The Azure module will read the environment variables `AZURE_STORAGE_ACCOUNT` and `AZURE_STORAGE_ACCESS_KEY`, or `AZURE_STORAGE_CONNECTION_STRING`, for information required to connect to your Azure storage account. If these environment variables are not set, you must specify the account information when calling **createBlobService**.

Store and retrieve blobs

Create a container

The **BlobService** object lets you work with containers and blobs. The following code creates a **BlobService** object. Add the following near the top of your code:

```
var blobSvc = azure.createBlobService();
```

NOTE

You can access a blob anonymously by using createBlobServiceAnonymous and providing the host address. For example, use `var blobSvc = azure.createBlobServiceAnonymous('https://myblob.blob.core.windows.net/');`.

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URIs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.

2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.

3. All letters in a container name must be lowercase.

4. Container names must be from 3 through 63 characters long.

NOTE

Important: the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

To create a new container, use **createContainerIfNotExists**. The following code example creates a new container named 'mycontainer':

```
blobSvc.createContainerIfNotExists('mycontainer', function(error, result, response){
    if(!error){
      // Container exists and is private
    }
});
```

If the container is newly created, `result.created` is true. If the container already exists, `result.created` is `false. response` contains information about the operation, including the ETag information for the container.

Container security

By default, new containers are private and you cannot access them anonymously. To make the container public so that you can access it anonymously, you can set the container's access level to **blob** or **container**.

- **Blob** - allows anonymous read access to blob content and metadata within this container, but not to container metadata such as listing all blobs within a container
- **Container** - allows anonymous read access to blob content and metadata as well as container metadata

The following code example demonstrates setting the access level to **blob**:

```
blobSvc.createContainerIfNotExists('mycontainer', {publicAccessLevel : 'blob'}, function(error, result, response){
    if(!error){
      // Container exists and allows
      // anonymous read access to blob
      // content and metadata within this container
    }
});
```

Alternatively, you can modify the access level of a container by using **setContainerAcl** to specify the access level. The following code example changes the access level to container:

```
blobSvc.setContainerAcl('mycontainer', null /* signedIdentifiers */, {publicAccessLevel : 'container'} /* publicAc-
cessLevel*/, function(error, result, response){
  if(!error){
    // Container access level set to 'container'
  }
});
```

The result contains information about the operation, including the current **ETag** for the container.

Filters

You can apply optional filtering operations to operations performed using BlobService. Filtering operations can include logging, automatically retrying, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After doing its preprocessing on the request options, the method needs to call "next", passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the *returnObject* (the response from the request to the server), the callback needs to either invoke next if it exists to continue processing other filters or simply invoke *finalCallback* to end the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, **ExponentialRetryPolicyFilter** and **LinearRetryPolicyFilter**. The following creates a BlobService object that uses the **ExponentialRetryPolicyFilter**:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var blobSvc = azure.createBlobService().withFilter(retryOperations);
```

Upload a blob into a container

There are three types of blobs: block blobs, append blobs and page blobs. Block blobs allow you to more efficiently upload large data. Append blobs are optimized for append operations. Page blobs are optimized for Virtual Machine disk storage, and are not covered here.

Block blobs

Use the following to upload data to a block blob:

- **createBlockBlobFromLocalFile** - creates a new block blob and uploads the contents of a file
- **createBlockBlobFromStream** - creates a new block blob and uploads the contents of a stream
- **createBlockBlobFromText** - creates a new block blob and uploads the contents of a string
- **createWriteStreamToBlockBlob** - creates a new block blob and then provides a stream to write to it

The following code example uploads the contents of the **test.txt** file into **myblob.**

```
blobSvc.createBlockBlobFromLocalFile('mycontainer','myblob', 'test.txt', function(error, result, response){
  if(!error){
    // file uploaded
  }
});
```

Append blobs

Use the following to upload data to a new append blob:

- **createAppendBlobFromLocalFile** - creates a new append blob and uploads the contents of a file
- **createAppendBlobFromStream** - creates a new append blob and uploads the contents of a stream
- **createAppendBlobFromText** - creates a new append blob and uploads the contents of a string
- **createWriteStreamToNewApendBlob** - creates a new append blob and then provides a stream to write to it

The following code example uploads the contents of the **test.txt** file into **myappendblob.**

```
blobSvc.createAppendBlobFromLocalFile('mycontainer', 'myappendblob', 'test.txt', function(error, result, response){
  if(!error){
    // file uploaded
  }
});
```

Use the following to upload data to a new append blob:

- **appendFromLocalFile** - creates a new append blob and uploads the contents of a file
- **appendFromStream** - creates a new append blob and uploads the contents of a stream
- **appendFromText** - creates a new append blob and uploads the contents of a string
- **appendBlockFromStream** - creates a new append blob and then provides a stream to write to it
- **createWriteStreamToNewApendBlob** - creates a new append blob and then provides a stream to write to it

NOTE

appendFromXXX APIs will do some client-side validation to fail fast to avoid unnecessary server calls. appendBlockFromXXX won't.

The following code example uploads the contents of the **test.txt** file into **myappendblob**.

```
blobSvc.appendFromText('mycontainer', 'myappendblob', 'text to be appended', function(error, result, response){
  if(!error){
    // text appended
  }
});
```

Download blobs

Use the following to download data from a blob:

- **getBlobToLocalFile** - writes the blob contents to file
- **getBlobToStream** - writes the blob contents to a stream
- **getBlobToText** - writes the blob contents to a string
- **createReadStream** - provides a stream to read from the blob

The following code example demonstrates using **getBlobToStream** to download the contents of the **myblob** blob and store it to the **output.txt** file by using a stream:

```
var fs = require('fs');
blobSvc.getBlobToStream('mycontainer','myblob', fs.createWriteStream('output.txt'), function(error, result, response){
  if(!error){
    // blob retrieved
  }
});
```

The `result` contains information about the blob, including **ETag** information.

Delete a blob

Finally, to delete a blob, call **deleteBlob**. The following code example deletes the blob named **myblob**.

```
blobSvc.deleteBlob(containerName,'myblob', function(error, response){
  if(!error){
    // Blob has been deleted
  }
});
```

Manage blob access

Concurrent access

To support concurrent access to a blob from multiple clients or multiple process instances, you can use **ETags** or **leases**.

- **Etag** - provides a way to detect that the blob or container has been modified by another process
- **Lease** - provides a way to obtain exclusive, renewable, write or delete access to a blob for a period of time.

ETags

Use ETags if you need to allow multiple clients or instances to write to the block Blob or page Blob simultaneously. The ETag allows you to determine if the container or blob was modified since you initially read or created it, which allows you to avoid overwriting changes committed by another client or process.

You can set ETag conditions by using the optional `options.accessConditions` parameter. The following code example only uploads the test.txt file if the blob already exists and has the ETag value contained by `etagToMatch.`

```
blobSvc.createBlockBlobFromLocalFile('mycontainer', 'myblob', 'test.txt', { accessConditions: { EtagMatch: etagToMatch} },
function(error, result, response){
    if(!error){
    // file uploaded
  }
});
```

When you're using ETags, the general pattern is:

1. Obtain the ETag as the result of a create, list, or get operation.
2. Perform an action, checking that the ETag value has not been modified.

If the value was modified, it indicates that another client or instance modified the blob or container since you obtained the ETag value.

Lease

You can acquire a new lease by using the **acquireLease** method, specifying the blob or container that you wish to obtain a lease on. For example, the following code acquires a lease on **myblob**.

```
blobSvc.acquireLease('mycontainer', 'myblob', function(error, result, response){
  if(!error) {
    console.log('leaseId: ' + result.id);
  }
});
```

Subsequent operations on myblob must provide the `options.leaseId` parameter. The lease ID is returned as `result.id` from acquireLease.

NOTE

By default, the lease duration is infinite. You can specify a non-infinite duration (between 15 and 60 seconds) by providing the `options.leaseDuration` parameter.

To remove a lease, use **releaseLease**. To break a lease, but prevent others from obtaining a new lease until the original duration has expired, use **breakLease**.

Work with shared access signatures

Shared access signatures (SAS) are a secure way to provide granular access to blobs and containers without providing your storage account name or keys. You can use shared access signatures to provide limited access to your data, such as allowing a mobile app to access blobs.

> **NOTE**
>
> While you can also allow anonymous access to blobs, shared access signatures allow you to provide more controlled access, as you must generate the SAS.

A trusted application such as a cloud-based service generates shared access signatures using the **generateSharedAccessSignature** of the **BlobService**, and provides it to an untrusted or semi-trusted application such as a mobile app. Shared access signatures are generated using a policy, which describes the start and end dates during which the shared access signatures are valid, as well as the access level granted to the shared access signatures holder.

The following code example generates a new shared access policy that allows the shared access signatures holder to perform read operations on the **myblob** blob, and expires 100 minutes after the time it is created.

```
var startDate = new Date();
var expiryDate = new Date(startDate);
expiryDate.setMinutes(startDate.getMinutes() + 100);
startDate.setMinutes(startDate.getMinutes() - 100);

var sharedAccessPolicy = {
  AccessPolicy: {
    Permissions: azure.BlobUtilities.SharedAccessPermissions.READ,
    Start: startDate,
    Expiry: expiryDate
  },
};

var blobSAS = blobSvc.generateSharedAccessSignature('mycontainer', 'myblob', sharedAccessPolicy);
var host = blobSvc.host;
```

A trusted application such as a cloud-based service generates shared access signatures using the **generateSharedAccessSignature** of the **BlobService**, and provides it to an untrusted or semi-trusted application such as a mobile app. Shared access signatures are generated using a policy, which describes the start and end dates during which the shared access signatures are valid, as well as the access level granted to the shared access signatures holder.

The following code example generates a new shared access policy that allows the shared access signatures holder to perform read operations on the **myblob** blob, and expires 100 minutes after the time it is created.

You must also provide the host information, as it is required when the shared access signatures holder attempts to access the container.

The client application then uses shared access signatures with **BlobServiceWithSAS** to perform operations against the blob. The following gets information about **myblob**.

```
var sharedBlobSvc = azure.createBlobServiceWithSas(host, blobSAS);
sharedBlobSvc.getBlobProperties('mycontainer','myblob', function (error, result, response) {
  if(!error) {
    // retrieved info
  }
});
```

Since the shared access signatures were generated with read-only access, if an attempt is made to modify the blob, an error will be returned.

Access control lists

You can also use an access control list (ACL) to set the access policy for SAS. This is useful if you wish to allow multiple clients to access a container but provide different access policies for each client.

An ACL is implemented using an array of access policies, with an ID associated with each policy. The following code example defines two policies, one for 'user1' and one for 'user2':

```
var sharedAccessPolicy = {
  user1: {
    Permissions: azure.BlobUtilities.SharedAccessPermissions.READ,
    Start: startDate,
    Expiry: expiryDate
  },
  user2: {
    Permissions: azure.BlobUtilities.SharedAccessPermissions.WRITE,
    Start: startDate,
    Expiry: expiryDate
  }
};
```

The following code example gets the current ACL for **mycontainer**, and then adds the new policies using **setBlobAcl**. This approach allows:

```
var extend = require('extend');
blobSvc.getBlobAcl('mycontainer', function(error, result, response) {
  if(!error){
    var newSignedIdentifiers = extend(true, result.signedIdentifiers, sharedAccessPolicy);
    blobSvc.setBlobAcl('mycontainer', newSignedIdentifiers, function(error, result, response){
      if(!error){
        // ACL set
      }
    });
  }
});
```

Once the ACL is set, you can then create shared access signatures based on the ID for a policy. The following code example creates new shared access signatures for 'user2':

```
blobSAS = blobSvc.generateSharedAccessSignature('mycontainer', { Id: 'user2' });
```

Next steps

For more information related to this chapter, see the following resources.

- Azure Storage SDK for Node API Reference
- Azure Storage SDK for Node repository on GitHub
- Transfer data with the AzCopy command-line utility
- For easy-to-use end-to-end Azure Storage code samples that you can download and run, check out our list of Azure Storage Samples

In this e-book you have learned how to deploy a cloud-native application using Node.js.

In the future, new trends, user demands and capabilities will continue to create even more opportunities to enhance your applications. Take advantage of the resources and guidance below to stay up to date on what you can do with Azure and Node.js:

Documentation
Visit the Azure for Node.js Developers documentation center to find the latest technical guidance, developer tools, code samples and quickstart guides to help you build your next Node.js project on Azure.

Free Training
Explore our selection of free online training courses for Node.js development on Azure from Pluralsight.

Free Training
Subscribe to our Microsoft+Open Source blog and follow @OpenAtMicrosoft on Twitter to keep in touch and stay up to date on the latest open source news and updates at Microsoft.