

Cosmos DB con NodeJS

Introducción

Este documento servirá como representación de las características generales y consideraciones de la api de nóminas en desarrollo apartir de los conocimientos y aplicación práctica de bases de datos NoSQL, visualizadores de datos y procesamiento de datos.

En la actualidad la cantidad de información en el mundo crece con gran velocidad, es por esto que su almacenamiento se ha visto limitado en cuanto a características como el volumen, velocidad y escalabilidad de la información, siendo esto el motivo principal para problemas relevantes como la pérdida de información y sobrecostos.

Teniendo en cuenta lo anterior, se puede afirmar que la información tiene un enorme poder dentro de muchas áreas, específicamente en la toma de decisiones en cualquier organización, puesto que proporciona una ventaja competitiva o una oportunidad de negocio.

Una posible solución al problema de almacenamiento para grandes volúmenes de datos y su consulta de manera rápida y eficiente es la incorporación de nuevos enfoques tecnológicos en cuanto al almacenamiento, tal enfoque es conocido como bases de datos NoSQL.

ComosDB

Este documento presenta una descripción general de todos los puntos que expone el desarrollo de un API Rest. Las aplicaciones necesitan comunicarse con la base de datos de Azure Cosmos DB, para ello, la creación de API REST es necesaria. Una vez generada, se puede comenzar a enviar inmediatamente llamadas a la API REST desde las aplicaciones cliente, agregar lógica comercial al principio y al final de su proceso de solicitud y respuesta, y configurar relaciones de esquemas personalizados.

Crear cuenta de Azure Cosmos DB

Se debe crear una cuenta de Azure Cosmos DB. En el menú de Azure Portal o en la **página de inicio** , seleccione **Crear un recurso** .

1. En la página **Nuevo** , busque y seleccione **Azure Cosmos DB** .
2. En la página de **Azure Cosmos DB** , seleccione **Crear** .
3. En la página **Crear cuenta de Azure Cosmos DB** , ingrese la configuración básica para la nueva cuenta de Azure Cosmos.

Configuración	Valor	Descripción
Suscripción	Nombre de la suscripción	Seleccione la suscripción de Azure que desea usar para esta cuenta de Azure Cosmos.
Grupo de recursos	Nombre del grupo de recursos	Seleccione un grupo de recursos o seleccione Crear nuevo , luego ingrese un nombre único para el nuevo grupo de recursos.
Nombre de la cuenta	Un nombre único	<p>Ingrese un nombre para identificar su cuenta de Azure Cosmos. Debido a que <i>documents.azure.com</i> se agrega al nombre que proporcionaste para crear tu URI, usa un nombre único.</p> <p>El nombre solo puede contener letras minúsculas, números y el carácter de guión (-). Debe tener entre 3 y 31 caracteres de longitud.</p>

Configuración	Valor	Descripción
API	El tipo de cuenta para crear	<p>Seleccione Core (SQL) para crear una base de datos de documentos y una consulta utilizando la sintaxis SQL.</p> <p>La API determina el tipo de cuenta a crear. Azure Cosmos DB proporciona cinco API: Core (SQL) y MongoDB para datos de documentos, Gremlin para datos de gráficos, Azure Table y Cassandra. Actualmente, debe crear una cuenta separada para cada API.</p> <p>Obtenga más información sobre la API de SQL .</p>
Localización	La región más cercana a sus usuarios	<p>Seleccione una ubicación geográfica para alojar su cuenta de Azure Cosmos DB. Utilice la ubicación más cercana a sus usuarios para brindarles el acceso más rápido a los datos.</p>
Modo de capacidad	Rendimiento aprovisionado o sin servidor	<p>Seleccione Rendimiento aprovisionado para crear una cuenta en modo de rendimiento aprovisionado . Seleccione Sin servidor para crear una cuenta en modo sin servidor .</p>
Aplicar el descuento de nivel gratuito de Azure Cosmos DB	Aplicar o no aplicar	<p>Con el nivel gratuito de Azure Cosmos DB, obtendrá las primeras 1000 RU / sy 25 GB de almacenamiento de forma gratuita en una cuenta. Obtenga más información sobre el nivel gratuito .</p>

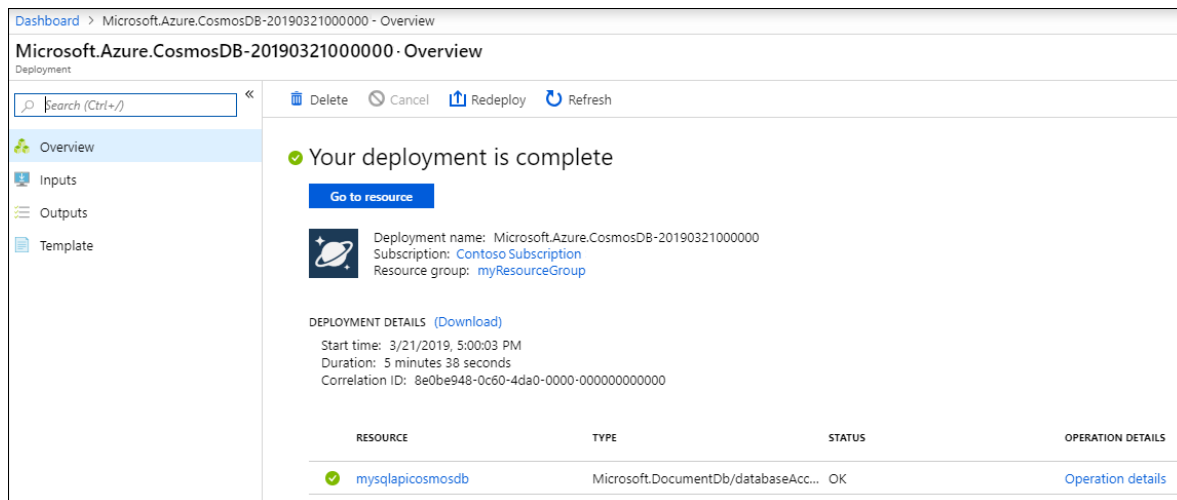
- En la pestaña **Distribución global** , configure los siguientes detalles. Puede dejar los valores predeterminados a los efectos de esta guía de inicio rápido:

Las siguientes opciones no están disponibles si selecciona Sin **servidor** como el **modo de capacidad** :

- Aplicar descuento de nivel gratuito
- Redundancia geográfica
- Escrituras multirregionales

Seleccione **Revisar + crear** .

5. Revise la configuración de la cuenta y luego seleccione **Crear** . Se necesitan unos minutos para crear la cuenta. Espere a que se muestre la página del portal . **Se completó la implementación** .



Dashboard > Microsoft.Azure.CosmosDB-20190321000000 - Overview

Microsoft.Azure.CosmosDB-20190321000000 - Overview

Deployment

Search (Ctrl+/)

Overview Inputs Outputs Template

✓ Your deployment is complete

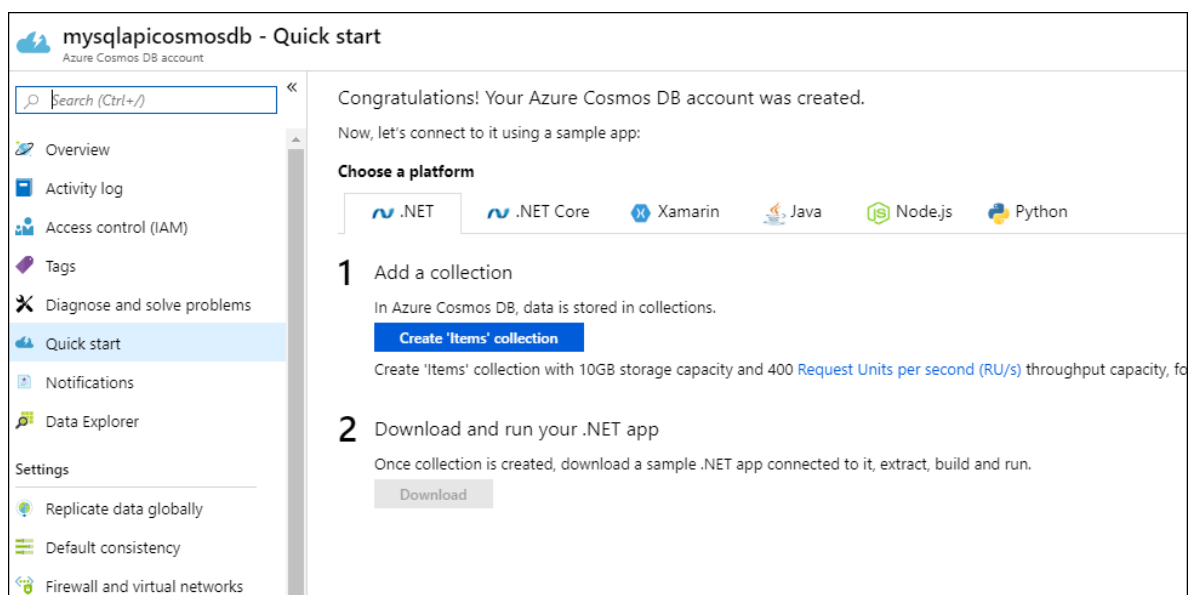
[Go to resource](#)

Deployment name: Microsoft.Azure.CosmosDB-20190321000000
Subscription: [Contoso Subscription](#)
Resource group: [myResourceGroup](#)

DEPLOYMENT DETAILS [\(Download\)](#)
Start time: 3/21/2019, 5:00:03 PM
Duration: 5 minutes 38 seconds
Correlation ID: 8e0be948-0c60-4da0-0000-000000000000

RESOURCE	TYPE	STATUS	OPERATION DETAILS
✓ mysqlapicosmosdb	Microsoft.DocumentDb/databaseAcc...	OK	Operation details

6. Seleccione **Ir al recurso** para ir a la página de la cuenta de Azure Cosmos DB.



mysqlapicosmosdb - Quick start

Azure Cosmos DB account

Search (Ctrl+/)

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Quick start Notifications Data Explorer Settings Replicate data globally Default consistency Firewall and virtual networks

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

Choose a platform

.NET .NET Core Xamarin Java Node.js Python

1 Add a collection

In Azure Cosmos DB, data is stored in collections.

[Create 'Items' collection](#)

Create 'Items' collection with 10GB storage capacity and 400 [Request Units per second \(RU/s\)](#) throughput capacity, for

2 Download and run your .NET app

Once collection is created, download a sample .NET app connected to it, extract, build and run.

[Download](#)

Configura tu aplicación Node.js

Antes de comenzar a escribir código para crear la aplicación, puede crear el marco para su aplicación. Ejecute los siguientes pasos para configurar su aplicación Node.js que tiene el código de marco:

1. Abre la terminal.
2. Busca la carpeta o directorio donde le gustaría guardar su aplicación Node.js.
3. Cree archivos JavaScript vacíos con los siguientes comandos:
 - Ventanas:
 - `fsutil file createnew app.js 0`
 - `fsutil file createnew config.js 0`
 - `md data`
 - `fsutil file createnew data\databaseContext.js 0`
 - Linux / OS X:
 - `touch app.js`
 - `touch config.js`
 - `mkdir data`
 - `touch data/databaseContext.js`
4. Crea e inicializa un `package.json` archivo. Utilice el siguiente comando:
 - `npm init -y`
5. Instale el módulo `@azure/cosmos` a través de npm. Utilice el siguiente comando:
 - `npm install @azure/cosmos --save`

Establece las configuraciones de tu aplicación

Ahora que su aplicación existe, debe asegurarse de que pueda comunicarse con Azure Cosmos DB. Al actualizar algunas opciones de configuración, como se muestra en los siguientes pasos, puede configurar su aplicación para que se comunique con Azure Cosmos DB:

1. Abra el archivo `config.js` en su editor de texto favorito.
2. Copie y pegue el siguiente fragmento de código en el archivo `config.js` y establezca las propiedades `endpoint` y `key` en el URI del punto de conexión y la clave principal de Azure Cosmos DB. La base de datos, los nombres de

los contenedores se establecen en **Tareas** y **Elementos** . La clave de partición que utilizará para esta aplicación es **/ categoría** .

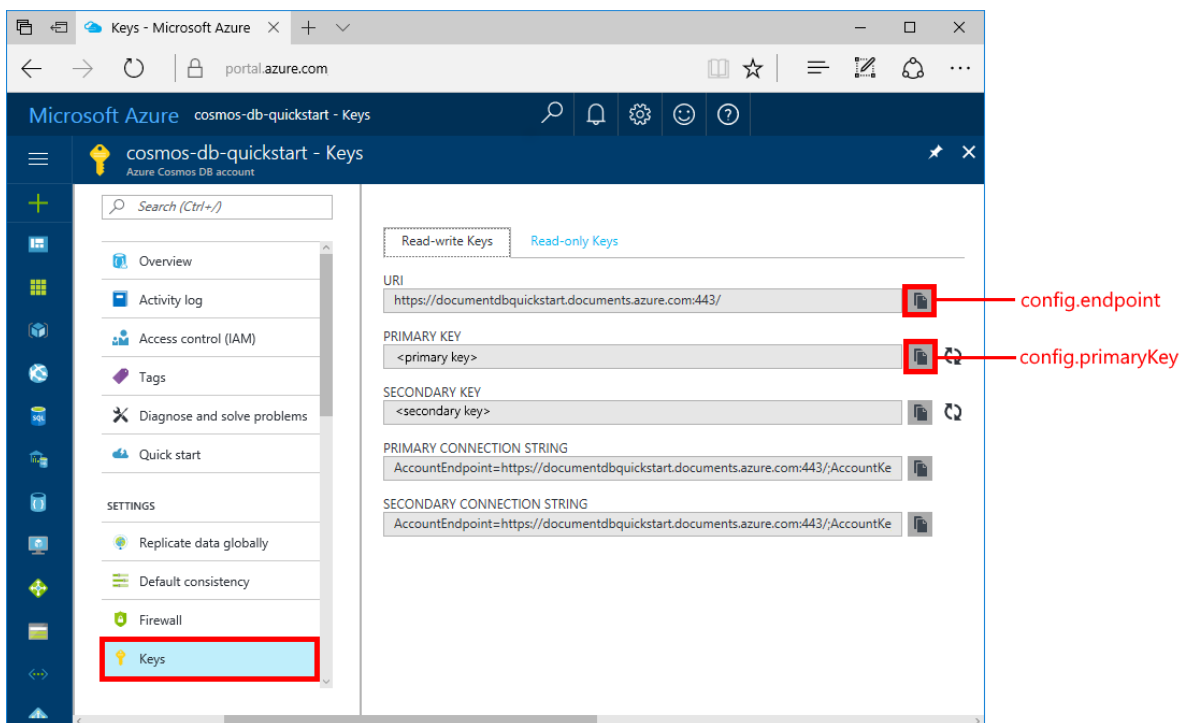
JavaScriptDupdo

```
// @ts-check
```

```
const config = {  
  endpoint: "<Your Azure Cosmos account URI>",  
  key: "<Your Azure Cosmos account key>",  
  databaseld: "Tasks",  
  containerId: "Items",  
  partitionKey: { kind: "Hash", paths: ["/category"] }  
};
```

```
module.exports = config;
```

Puede encontrar el punto de conexión y los detalles de la clave en el panel **Claves** de [Azure Portal](https://portal.azure.com) .



El SDK de JavaScript utiliza el *elemento* y el *contenedor* de términos genéricos . Un contenedor puede ser una colección, un gráfico o una tabla. Un elemento puede ser un documento, un borde / vértice o una fila, y es el contenido dentro de un contenedor. En el fragmento de código anterior, el código se usa para exportar el objeto de configuración, de modo que pueda hacer referencia a él dentro del archivo *app.js*.`module.exports = config;`

Crea una base de datos y un contenedor

1. Abra el archivo *databaseContext.js* en su editor de texto favorito.
2. Copie y pegue el siguiente código en el archivo *databaseContext.js* . Este código define una función que crea la base de datos "Tareas", "Elementos" y el contenedor si aún no existen en su cuenta de Azure Cosmos:

JavaScriptDupdo

```
const config = require("../config");
const CosmosClient = require("@azure/cosmos").CosmosClient;

/*
// This script ensures that the database is setup and populated correctly
*/

async function create(client, databaseId, containerId) {
  const partitionKey = config.partitionKey;

  /**
   * Create the database if it does not exist
   */
  const { database } = await client.databases.createIfNotExists({
    id: databaseId
  });
  console.log(`Created database:\n${database.id}\n`);

  /**
   * Create the container if it does not exist
```

```

*/

const { container } = await client
  .database(databaseId)
  .containers.createIfNotExists(
    { id: containerId, partitionKey },
    { offerThroughput: 400 }
  );

console.log(`Created container:\n${container.id}\n`);
}

```

```
module.exports = { create };
```

Una base de datos es el contenedor lógico de elementos divididos en contenedores. Puede crear una base de datos utilizando la `createIfNotExists` función o crear de la clase **Bases de datos** . Un contenedor consta de elementos que, en el caso de la API de SQL, son documentos JSON. Puede crear un contenedor utilizando la `createIfNotExists` función o `create` de la clase **Containers** . Después de crear un contenedor, puede almacenar y consultar los datos.

Importar la configuración

1. Abra el archivo `app.js` en su editor de texto favorito.
2. Copie y pegue el código a continuación para importar el `@azure/cosmos` módulo, la configuración y el `databaseContext` que definió en los pasos anteriores.

JavaScriptDuplicado

```

const CosmosClient = require("@azure/cosmos").CosmosClient;

const config = require("./config");

const dbContext = require("./data/databaseContext");

```

Conéctese a la cuenta de Azure Cosmos

En el archivo `app.js` , copie y pegue el siguiente código para usar el punto final y la clave previamente guardados para crear un nuevo objeto `CosmosClient`.

JavaScriptDuplicado


```
const { endpoint, key, databaseld, containerId } = config;
```

```
const client = new CosmosClient({ endpoint, key });
```

```
const database = client.database(databaseld);
```

```
const container = database.container(containerId);
```

```
// Make sure Tasks database is already setup. If not, create it.
```

```
await dbContext.create(client, databaseld, containerId);
```

Nota

Si se conecta al **emulador de Cosmos DB**, desactive la verificación TLS para su proceso de nodo:

JavaScriptDupdo

```
process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";
```

```
const client = new CosmosClient({ endpoint, key });
```

Ahora que tiene el código para inicializar el cliente de Azure Cosmos DB, echemos un vistazo a cómo trabajar con los recursos de Azure Cosmos DB.

Elementos de consulta

Azure Cosmos DB admite consultas enriquecidas contra elementos JSON almacenados en cada contenedor. El siguiente código de muestra muestra una consulta que puede ejecutar en los elementos de su contenedor. Puede consultar los elementos mediante la función de consulta de la `Items` clase. Agregue el siguiente código al archivo *app.js* para consultar los elementos de su cuenta de Azure Cosmos:

JavaScriptDupdo

```
console.log(`Querying container: Items`);
```

```
// query to return all items
```

```
const querySpec = {
```

```
  query: "SELECT * from c"
```

```
};
```

```
// read all items in the Items container

const { resources: items } = await container.items

  .query(querySpec)

  .fetchAll();

items.forEach(item => {
  console.log(`${item.id} - ${item.description}`);
});
```

Crea un artículo

Se puede crear un elemento utilizando la función de creación de la Itemsclase. Cuando utiliza la API de SQL, los elementos se proyectan como documentos, que son contenido JSON definido por el usuario (arbitrario). En este tutorial, crea un nuevo elemento dentro de la base de datos de tareas.

1. En el archivo app.js, defina la definición del elemento:

JavaScriptDupdo

```
const newItem = {
  id: "3",
  category: "fun",
  name: "Cosmos DB",
  description: "Complete Cosmos DB Node.js Quickstart ⚡",
  isComplete: false
};
```

2. Agregue el siguiente código para crear el elemento previamente definido:

JavaScriptDupdo

```
/** Create new item
 * newItem is defined at the top of this file
 */

const { resource: createdItem } = await container.items.create(newItem);
```

```
console.log(`\r\nCreated      new      item:      ${createdItem.id}      -  
${createdItem.description}\r\n`);
```

Actualizar un artículo

Azure Cosmos DB admite la sustitución del contenido de los elementos. Copie y pegue el siguiente código en el archivo *app.js*. Este código obtiene un elemento del contenedor y actualiza el campo *isComplete* a verdadero.

JavaScriptDuplicado

```
/** Update item
```

```
 * Pull the id and partition key value from the newly created item.
```

```
 * Update the isComplete field to true.
```

```
 */
```

```
const { id, category } = createdItem;
```

```
createdItem.isComplete = true;
```

```
const { resource: updatedItem } = await container
```

```
  .item(id, category)
```

```
  .replace(createdItem);
```

```
console.log(`Updated item: ${updatedItem.id} - ${updatedItem.description}`);
```

```
console.log(`Updated isComplete to ${updatedItem.isComplete}\r\n`);
```

Eliminar un elemento

Azure Cosmos DB admite la eliminación de elementos JSON. El siguiente código muestra cómo obtener un artículo por su ID y eliminarlo. Copie y pegue el siguiente código en el archivo *app.js* :

JavaScriptDuplicado

```
/**
```

```
 * Delete item
```

```
 * Pass the id and partition key value to delete the item
```

```
*/  
  
const { resource: result } = await container.item(id, category).delete();  
console.log(`Deleted item with id: ${id}`);
```

Ejecute su aplicación Node.js

En total, su código debería verse así:

JavaScriptDupdo

```
// @ts-check  
  
// <ImportConfiguration>  
  
const CosmosClient = require("@azure/cosmos").CosmosClient;  
const config = require("./config");  
const dbContext = require("./data/databaseContext");  
  
// </ImportConfiguration>  
  
  
// <DefineNewItem>  
const newItem = {  
  id: "3",  
  category: "fun",  
  name: "Cosmos DB",  
  description: "Complete Cosmos DB Node.js Quickstart ⚡",  
  isComplete: false  
};  
  
// </DefineNewItem>  
  
  
async function main() {  
  
  // <CreateClientObjectDatabaseContainer>  
  const { endpoint, key, databaseId, containerId } = config;
```

```

const client = new CosmosClient({ endpoint, key });

const database = client.database(databaseld);
const container = database.container(containerId);

// Make sure Tasks database is already setup. If not, create it.
await dbContext.create(client, databaseld, containerId);
// </CreateClientObjectDatabaseContainer>

try {
  // <QueryItems>
  console.log(`Querying container: Items`);

  // query to return all items
  const querySpec = {
    query: "SELECT * from c"
  };

  // read all items in the Items container
  const { resources: items } = await container.items
    .query(querySpec)
    .fetchAll();

  items.forEach(item => {
    console.log(`${item.id} - ${item.description}`);
  });
  // </QueryItems>

```

```

// <CreateItem>
/** Create new item
 * newItem is defined at the top of this file
 */

const { resource: createdItem } = await container.items.create(newItem);

console.log(`\r\nCreated      new      item:      ${createdItem.id}      -
${createdItem.description}\r\n`);
// </CreateItem>

// <UpdateItem>
/** Update item
 * Pull the id and partition key value from the newly created item.
 * Update the isComplete field to true.
 */

const { id, category } = createdItem;

createdItem.isComplete = true;

const { resource: updatedItem } = await container
    .item(id, category)
    .replace(createdItem);

console.log(` Updated item: ${updatedItem.id} - ${updatedItem.description}`);
console.log(` Updated isComplete to ${updatedItem.isComplete}\r\n`);
// </UpdateItem>

// <DeleteItem>
/**

```

```

    * Delete item
    * Pass the id and partition key value to delete the item
    */
    const { resource: result } = await container.item(id, category).delete();
    console.log(`Deleted item with id: ${id}`);
    // </DeleteItem>

  } catch (err) {
    console.log(err.message);
  }
}

main();

```

En su terminal, ubique su app.js archivo y ejecute el comando:

```
node app.js
```

Debería ver el resultado de su aplicación de introducción. La salida debe coincidir con el texto de ejemplo a continuación.

Conclusión

Como cualquier desarrollo, este tipo de arquitectura y diseño tiene sus beneficios e inconvenientes, pero podría destacar:

1. No es necesario realizar mantenimiento de los servidores donde se tienen instalados programas y aplicaciones. El código se ejecuta en un contenedor temporal por lo que ya no es necesario instalar software, gestionar puertos de acceso o estar pendiente de las actualizaciones.
2. Se puede realizar un escalamiento de manera horizontal tanto como se requiera. Es posible añadir todos los clusters, balanceo de cargas etc, conforme se necesite.

3. Relacionado a los costos, solamente se va a pagar por el tiempo que se estén utilizando los procesos.

4. Las funciones que se utilicen, es posible integrarlas con el resto de servicios que ofrece la plataforma, como son logging, virtualización o endpoints.