Introduction to Spring Data Azure Cosmos DB

Last updated: January 8, 2024

Written

(/)

Reviewed by: Rui Vilao (https://www.baeldung.com/editor/rui-vilao)

Cloud (https://www.baeldung.com/category/cloud)

Spring Data (https://www.baeldung.com/category/persistence/spring-persistence/spring-data)

Azure (https://www.baeldung.com/tag/azure)

1. Overview

In this tutorial, we'll learn about Azure Cosmos DB and how we can interact with it using Spring Data.

2. Azure Cosmos DB

Azure Cosmos DB (https://github.com/Azure/azure-sdk-for-java/tree/master/sdk/cosmos/azure-cosmos) is Microsoft's globally distributed database service.

It's a NoSQL database, which provides comprehensive service level agreements for throughput, latency, availability, and consistency guarantees. Also, it assures 99.999% availability (https://docs.microsoft.com/en-us/azure/cosmos-db/introduction#guaranteed-low-latency-at-99th-percentile-worldwide) for both reads and writes.

Azure Cosmos DB does not give only two consistency choices, i.e. either consistent or not consistent. Instead, **we get five consistency choices: *strong*, *bounded staleness*, *session*, *consistent prefix*, and *eventual*.**

We can elastically scale both the throughput and storage of Azure Cosmos DB.

Additionally, it's available in all Azure regions and offers turnkey global distribution, as we can replicate our data in any Azure region just by clicking a button. This helps us have our data closer to our users so we can serve their requests faster.

It's schema-agnostic as it has no schema. Furthermore, we don't need to do any index management for Azure Cosmos Db. It automatically does the indexing of data for us.

We can work with Azure CosmosDb using different standard APIs such as SQL, MongoDB, Cassandra, etc.

3. Spring Data Azure Cosmos DB

Microsoft also provides a module that allows us to work with Cosmos DB using Spring Data. In the next section, we'll see how we can use Azure Cosmos DB in a Spring Boot application.

In our example, we'll create a Spring web application that stores a product entity in an Azure Cosmos database and performs basic CRUD operations on it. First, we need to configure an account and database in the Azure portal, following the instructions in the documentation (https://docs.microsoft.com/en-us/azure/cosmos-db/create-cosmosdb-resources-portal).

If we don't want to create an account on the Azure portal, Azure also provides the Azure Cosmos Emulator. Even though this doesn't contain all the functionalities of Azure Cosmos Service and there are some differences, we can use it for local development and testing.

We can use the Emulator in our local environment in two ways: either by downloading the Azure Cosmos Emulator on our machine or running the Emulator (https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator#running-on-docker) on Docker for Windows.

We'll choose the option to run it on Docker for Windows. Let's pull the Docker image by running the following command:

```
docker pull mcr.microsoft.com/cosmosdb/windows/azure-cosmos-emulator
```

Then we can run the Docker image and start the container by running the following commands:

```
md $env:LOCALAPPDATA\CosmosDBEmulator\bind-mount 2>null
docker run --name azure-cosmosdb-emulator --memory 2GB --mount
"type=bind,source=$env:LOCALAPPDATA\CosmosDBEmulator\bind-mount,destination=C:\CosmosDB.Emulator\bind-
mount"
--interactive --tty -p 8081:8081 -p 8900:8900 -p 8901:8901 -p 8902:8902 -p 10250:10250
-p 10251:10251 -p 10252:10252 -p 10253:10253 -p 10254:10254 -p 10255:10255 -p 10256:10256 -p 10350:10350
mcr.microsoft.com/cosmosdb/windows/azure-cosmos-emulator
```


Once we've configured the Azure Cosmos DB account and database, we can move on to the certificate configurations.

3.1. Configure Azure Cosmos Emulator Certificate

The emulator internally uses a self-signed certificate that must be imported to our keystore. Alternatively, we can disable the TLS/SSL validation, which is not recommended. The local installation on Windows imports the certificates for us, and we don't have to perform this step. But, since we are using docker containers, we need to manually import the certificates into our Java keystore using keytool (/keytool-intro).

To obtain the certificate, we can make a curl to the docker container, which exposes an endpoint to the 8081 port:

```
curl -k https://localhost:8081/_explorer/emulator.pem > ~/emulatorcert.crt
```

Now we can import the certificate (/java-import-cer-certificate-into-keystore) in our keystore called *cacerts*. If the keystore doesn't exist, it will be created for use:

```
keytool -importcert -alias cacerts -keystore cacerts -file ~/emulatorcert.crt
```

We can also do it programmatically (/java-keystore) inside our application. But, for brevity, we won't show that and move on to configuring the Spring application.

4. Using Azure Cosmos DB in Spring

4.1. Configuring Spring Data Azure Cosmos DB with Spring

We start by adding the spring-data-cosmosdb (https://mvnrepository.com/artifact/com.microsoft.azure/spring-data-cosmosdb/2.3.0) dependency in our *pom.xml*:


```
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>spring-data-cosmosdb</artifactId>
    <version>2.3.0</version>
</dependency>
```

To access Azure Cosmos DB from our Spring application, we'll need the URI of our database, its access keys (https://docs.microsoft.com/en-us/azure/cosmos-db/secure-access-to-data) and the database name. Then we'll add the connection properties in our *application.properties*:

```
azure.cosmosdb.uri=cosmodb-uri
azure.cosmosdb.key=cosmodb-primary-key
azure.cosmosdb.secondaryKey=cosmodb-secondary-key
azure.cosmosdb.database=cosmodb-name
```

We can find the values of the above properties from the Azure portal. The URI, primary key, and secondary key will be available in the keys section of our Azure Cosmos DB in the Azure portal.

To connect to Azure Cosmos DB from our application, we need to create a client. For that, **we need to extend *AbstractCosmosConfiguration* class in our configuration class and add the *@EnableCosmosRepositories* annotation.**

This annotation will scan for interfaces that extend Spring Data's repository interfaces in the specified package.

We also need to **configure a bean of type *CosmosDBConfig***.

```
@Configuration
@EnableCosmosRepositories(basePackages = "com.baeldung.spring.data.cosmosdb.repository")
public class AzureCosmosDbConfiguration extends AbstractCosmosConfiguration {

    @Value("${azure.cosmosdb.uri}")
    private String uri;

    @Value("${azure.cosmosdb.key}")
    private String key;

    @Value("${azure.cosmosdb.database}")
    private String dbName;

    private CosmosKeyCredential cosmosKeyCredential;

    @Bean
    public CosmosDBConfig getConfig() {
        this.cosmosKeyCredential = new CosmosKeyCredential(key);
        CosmosDBConfig cosmosdbConfig = CosmosDBConfig.builder(uri, this.cosmosKeyCredential, dbName)
            .build();
        return cosmosdbConfig;
    }
}
```


4.2. Creating an Entity for Azure Cosmos DB

To interact with Azure Cosmos DB, we make use of entities. So, let's create an entity that we will store in Azure Cosmos DB. To make our *Product* class an entity, **we'll use the *@Document* annotation:**

```
@Document(collection = "products")
public class Product {

    @Id
    private String productid;

    private String productName;

    private double price;

    @PartitionKey
    private String productCategory;
}
```


In this example, **we've used the *collection* attribute with the value *products* to indicate this will be the name of our container in the database**. If we don't provide any value for the *collection* parameter, then the class name will be used as the container name in the database.

We've also defined an id for our document. We can either create a field with the name *id* in our class or annotate a field with the *@Id* annotation. Here we have used the *productid* field as the document id.

We can logically partition our data in our container by using a partition key by annotating a field with *@PartitionKey.* In our class, we've used the *productCategory* field as the partition key.

By default, the indexing policy is defined by Azure, but we can also customize it by using *@DocumentIndexingPolicy* annotation on our entity Class.

We can also enable Optimistic locking for our entity container by creating a field named *_etag* and annotating it with *@Version.*

4.3. Defining the Repository

Now **let's create a *ProductRepository* interface that extends *CosmosRepository***. Using this interface, we can perform CRUD operations on our Azure Cosmos DB:

```
@Repository
public interface ProductRepository extends CosmosRepository<Product, String> {
    List findByProductName(String productName);

}
```

As we can see, this is defined in a similar way to other Spring Data modules.

4.4. Testing the Connection

Now we can create a Junit Test to save a *Product* entity in Azure Cosmos DB using our *ProductRepository*:


```
@SpringBootTest
public class AzureCosmosDbApplicationManualTest {

    @Autowired
    ProductRepository productRepository;

    @Test
    public void givenProductIsCreated_whenCallFindById_thenProductIsFound() {
        Product product = new Product();
        product.setProductid("1001");
        product.setProductCategory("Shirt");
        product.setPrice(110.0);
        product.setProductName("Blue Shirt");

        productRepository.save(product);
        Product retrievedProduct = productRepository.findById("1001", new PartitionKey("Shirt"))
            .orElse(null);
        Assert.notNull(retrievedProduct, "Retrieved Product is Null");
    }

}
```

By running this Junit test, we can test our connection with Azure Cosmos DB from our Spring application.

5. Conclusion

In this tutorial, we learned about Azure Cosmos DB. Further, we learned how to access Azure Cosmos DB from a Spring Boot application, how to create entities and configure a Repository by extending *CosmosRepository* to interact with it.

The code for the above example is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-cosmosdb).

2 COMMENTS

Oldest

View Comments

(/)

Start Here (/start-here) Spring Courses ▾ Java Courses ▾ Guides ▾ About ▾ (/feed)

COURSES

ALL COURSES (/COURSES/ALL-COURSES)

BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)

BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)

THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)

JACKSON JSON SERIES (/JACKSON)

APACHE HTTPCLIENT SERIES (/HTTPCLIENT-SERIES)

REST WITH SPRING SERIES (/REST-WITH-SPRING-SERIES)

SPRING PERSISTENCE SERIES (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

SPRING REACTIVE SERIES (/SPRING-REACTIVE-SERIES)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS/)

PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)

EBOOKS (/LIBRARY/)

FAQ (HTTPS://WWW.BAELDUNG.COM/LIBRARY/FAQ)

BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)

(/)

Start Here (/start-here) Spring Courses ▾ Java Courses ▾ Guides ▾ About ▾ (/feed)