Mark Heckler
Posted on Aug 19, 2022

Cosmos DB for Spring Developers, Part I: Using Cosmos DB as a SQL Database

Mark Heckler

Software developer & Principal Cloud Advocate for Java/JVM Languages @Microsoft, conference speaker, Java Champion, & Kotlin Dev Expert focused on developing production-ready software at velocity.

LOCATION
Central US

WORK
Software developer & Principal Cloud Developer Advocate for Java/JVM Languages at Microsoft

JOINED
Feb 9, 2020

More from Mark Heckler

Cosmos DB for Spring Developers, Part II: Using Cosmos DB as a

#springboot #cosmosdb #azure #spring

One of the key benefits of working with any Spring project, from Spring Framework to Spring Boot, Spring Data to Spring Cloud, is the developer-first focus each of those projects takes to delivering capabilities. This manifests in several ways, but the ways on which I want to focus with this series of articles are flexibility and scalability, especially with regard to database connectivity.

Delivering Value

The vast majority of systems and the applications that constitute them provide most of their value via data, whether it be via storage/retrieval, manipulation/analysis, transport/exchange, or some combination of these or other operations. As developers, data permeates aspects of everything we do, and yet we often don't give data considerations the same level of attention that we do our apps and the logic contained therein.

This may be due to the wealth of options we have - or assume, or even hope we have - at our disposal. But many times, it's largely due to the immediacy of the "demands

MongoDB Database

#springboot #cosmosdb #azure #mongodb

A Concise Guide to Using Multiple Azure Storage Accounts from a Single Spring Boot Application

#azure #springboot #azurestorage #java

of now": getting functionality developed, tested, and deployed, so we can move on to the next bit of needed functionality. There is no shortage of requirements after all; our customers and community are racing to provide their customers and community the functionality they need as well.

Spring Boot and Spring Data

Spring Boot and Spring Data enable us as developers to "get it done" effectively. Both are force multipliers, levers and fulcrums that take the heavy lifting of boilerplate and ceremony, replacing repetitious patterns to which we developers have grown numb with software-driven opinions. If you have to do the same handful of steps as setup and teardown (for example) with little or no variation every time you perform a set of basic operations, chances are those surrounding steps can be incorporated into an opinion, implemented as a convention one follows within Spring Boot (instead of a manual configuration), and streamlined away from the top level of developer concern. There must be ways to tweak, configure, and disable these steps for edge cases of course, but this is what Spring Boot does as a matter of course.

Spring Data adds convention over configuration and when used in a Spring Boot application, Boot's autoconfiguration makes accessing data via Spring Data straightforward via a variety of mechanisms: the Java Persistence API (JPA), Java Database Connectivity (JDBC), and various NoSQL database drivers for document, wide table, caching, in-memory, and graph databases to mention a few.

Let's talk about that Data

What Spring Data *doesn't* do is make determinations of scalability and suitability of various database implementations for a specific purpose or targeted use case. As technologists, that is for us to determine.

Sometimes we accurately anticipate functional and non-functional requirements. Sometimes scope and scalability requirements remain reasonably close to expectations. And sometimes, when amazing things happen and our systems gain far greater traction and a much wider user base than we could have anticipated, we must scale *everything* quickly and effectively, both our systems and the data on which they depend and use to provide such meaningful value. That's where Cosmos DB comes in.

Cosmos DB

Azure Cosmos DB is a planetary-scale data store that offers various ways and APIs to manage your critical data and provide access to it from anywhere at breathtaking speed. Spring Developers can leverage the power of Cosmos DB via Spring Boot starters, among other means, and those starters are built and maintained by Microsoft developers as natural complements to the opinions built into Spring Boot and Spring Data. Developing applications using Cosmos DB is easy and requires no adjustments to a developer's existing mental processes.

One powerful facet of Cosmos DB is its chameleon-like quality of fulfilling developers' needs for various APIs and storage mechanisms. Cosmos DB can operate and be interacted with as a SQL database, a MongoDB(tm) document database, a Cassandra(tm) wide table database, and more: each one at planetary scale and speed, each one with little or no adjustment to your applications needed.

For this article, I focus on the SQL engine and API for Cosmos DB, as leveraged from a Spring Boot application. Let's get started!

Initialization and configuration

I use two scripts (repository link at the bottom of this article) to prepare the Azure environment generally and Cosmos DB specifically for this project: one to initialize environment variables to be used, and one to create the Azure Cosmos DB target based upon those variables.

The script `CosmosInitEnv.sh` should be sourced by executing it in this manner from a terminal window:

```
source ./CosmosInitEnv.sh
```

This assumes that the script resides in the current (.) directory, but if not, you must specify the full path to the shell script. Sourcing the script sets the environment variables contained therein and ensures they remain part of the environment after the script finishes, rather than spawning a different environment to run the script and having all changes made in that process disappear upon completion.

To verify environment variables are set as expected, I like to check the environment using a command similar to the following:

```
env | sort
```

or

```
env | grep COSMOSDB
```

To create the Azure resources we will need for this project, we then source the `CosmosConfig.sh` script as follows:

```
source ./CosmosConfig.sh
```

Two things of note. First, this again assumes you are executing this script from the current (.) directory. Second, since we once again set environment variable values that we plan to use after the script finishes execution, we must source this script as well. Failure to do isn't catastrophic, but it will result in empty variables and will require manually executing the final two lines from `CosmosConfig.sh` in the terminal in order to proceed.

You can verify all environment variables are now set using this or similar command:

```
env | grep COSMOSDB_SQL
```

These three env vars are essential to the app we're about to create:

- COSMOSDB_SQL_URL
- COSMOSDB_SQL_KEY
- COSMOSDB_SQL_NAME

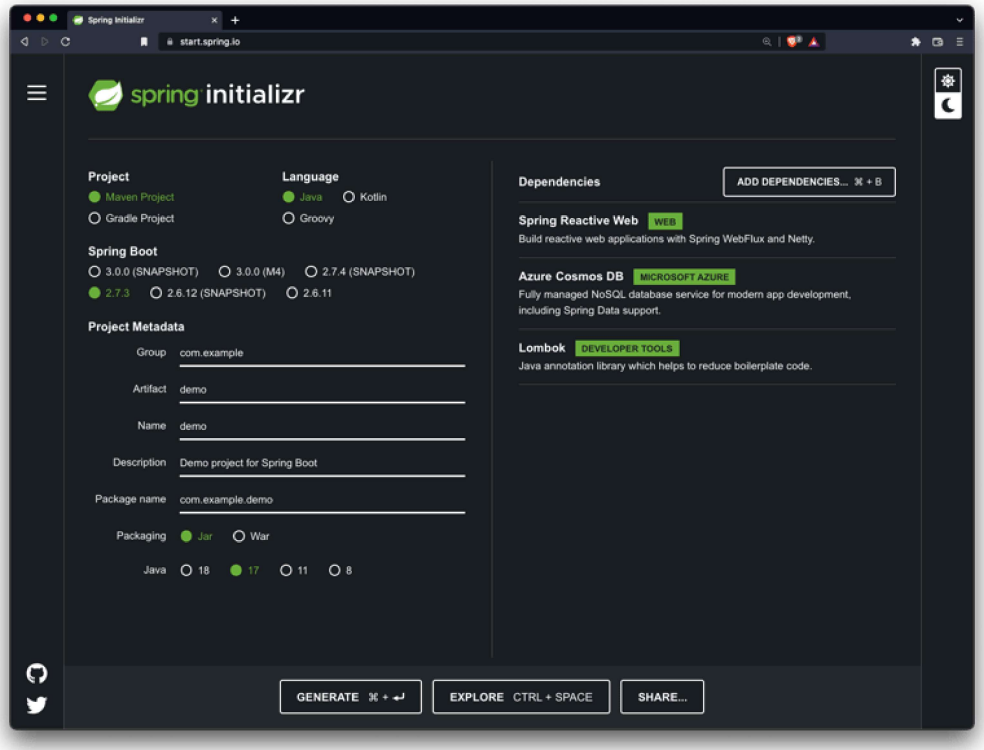
If all three variables' values are present, we can proceed to the next step and build an application.

Create a Spring Boot app

There are many ways to begin creating a Spring Boot application, but I prefer to begin with the Spring Initializr. Choose a meaningful artifact name, change the group name if you so choose, then select the dependencies as shown in the following screen capture.

Create and open the project

NOTE: All code is in a Github repository linked at the bottom of this article.



I use only three dependencies for this example:

- **Reactive Web** includes all dependencies to create web apps using reactive streams, non-blocking and/or imperative APIs
- **Azure Cosmos DB** enables your app to access Cosmos DB using the SQL API
- **Lombok** is a boilerplate-reducing library, useful for domain classes, logging, and more

Next, click the "Generate" button to generate the project structure and download the compressed project files (.zip). Once downloaded, go to the directory where the file was saved, decompress it, and open the project in the Integrated Development Environment (IDE) or text editor of your choice. I use IntelliJ IDEA and Visual Studio Code (VSCode) for nearly all my dev work, and for this article I'll open the project in IntelliJ by navigating (using the Mac Finder) into the expanded project directory and double-clicking on the Maven build file, *pom.xml*.

Once the project is loaded, you can verify that the dependencies chosen from the Spring Initializr are present within the project by opening the *pom.xml* file. There will be additional ones brought in for testing, etc., but these represent the three we selected before generating the project structure:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifact
</dependency>
<dependency>
```

```
        <groupId>com.azure.spring</groupId>
        <artifactId>spring-cloud-azure-starter-data-cosmos
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
```

Set required properties

Next, we must specify a few properties for our application to use in its initialization for Spring Boot's autoconfiguration to make the connection with the desired Cosmos DB instance. Open the *application.properties* file in the project (under *src/main/resources*) and add the following parameters:

```
spring.cloud.azure.cosmos.endpoint=${COSMOSDB_SQL_URL
spring.cloud.azure.cosmos.key=${COSMOSDB_SQL_KEY}
spring.cloud.azure.cosmos.database=${COSMOSDB_SQL_NAM
```

There are many ways for a Spring Boot app to ingest properties from its environment, but for this article, we'll simply use SpEL (Spring Expression Language) variables that map to underlying environment variables with values assigned by the scripts we sourced earlier from the command line.

Code the application

I'll start with a streamlined example for this article and plan to build out in follow-on posts.

The domain

First, I code a domain class. In this example, I create a `User` class with an `id`, `firstName`, `lastName`, and `address` member variables. This class and its properties are annotated thusly:

- `@Container(containerName = "data")` represents the container name within Cosmos DB

- `@Data` is a Lombok annotation that instructs the Lombok compile-time code generator to consider this a "data class" and generate accessors (getters) and mutators (setters) for each member variable, along with `equals()`, `hashCode()`, and `toString()` methods
- `@NoArgsConstructor` is a Lombok annotation that instructs Lombok to generate a zero-argument constructor
- `@RequiredArgsConstructor` is a Lombok annotation that instructs Lombok to generate a constructor with a parameter for each "required" member variable, as designated by the `@NonNull` member variable annotation
- `@Id` indicates which member variable corresponds to the underlying table's primary key
- `@GeneratedValue` specifies that the underlying data store will generate this value, i.e. it does not need to be provided by the application
- `@NonNull` is addressed under `@RequiredArgsConstructor`
- `@PartitionKey` corresponds to the key used by Cosmos DB to partition data stored for this domain entity for access optimization

NOTE: In this code, I use lastName as the partition key. This is a terrible choice in most cases, as it typically doesn't provide a meaningful and/or reasonably even grouping of entities for optimizing access. I use it in this simplified example for expediency and would strongly encourage the reader to research partitions with specific domains/data in mind. Choose wisely.

The repository

Spring Boot's autoconfiguration takes the power of Spring Data and amplifies it. By including a database driver in your classpath (including it as a dependency in your build file results in its inclusion for deployment) and `extend`ing a Spring Data-derived interface in your application code, Spring Boot's autoconfiguration creates the beans necessary to provide a proxy to the desired underlying datastore. In our case, this is all we need to provide foundational database capabilities:

```
interface UserRepository extends ReactiveCosmosReposi
```

In this single line of code, we are defining an interface called `UserRepository` that will inherit and potentially extend the capabilities of the `ReactiveCosmosRepository`, storing objects of type `User` with identifiers (IDs) of type `String`. Autoconfig does the rest.

NOTE: We can do more, of course, defining custom query methods and more. But for this example, the provided functionality is sufficient.

The API

Next, I define the Application Programming Interface (API) that provides the means and structure for external applications to interact with this service. Once again the focus is on simplicity, and I define only a single HTTP endpoint that provides a JSON listing (by default) of all Users upon request.

```
@RestController
```

```
@AllArgsConstructor
class CosmosSqlController {
    private final UserRepository repo;

    @GetMapping
    Flux<User> getAllUsers() {
        return repo.findAll();
    }
}
```

The `@RestController` annotation is provided by the Spring Framework and combines the functionality of `@Controller`, to respond to requests, and `@ResponseBody`, to make the resultant object(s) the response body itself rather than just providing access to the object(s) via a model variable, as is the typical MVC methodology.

The `@AllArgsConstructor` annotation instructs Lombok to create a constructor for the `CosmosSqlController` class with a parameter (and thus required argument) for every member variable. Since the only member variable is a `UserRepository`, Lombok generates a ctor with that single parameter.

NOTE: `@AllArgsConstructor` has the fortunate (or perilous) capability to update your constructor automatically if you simply add/remove member variables, so remember: With great power comes great responsibility. :)

The data

To create a bit of sample data, I create a Spring bean using the `@Component` annotation. This annotation instructs Spring Boot, upon application initialization, to create an instance of the annotated class and place that object (bean) into the application context, i.e. its Dependency Injection (DI) container. All beans in the DI container are managed by the Spring Boot application in terms of lifecycle and, of course, injection into other code as dependencies.

Once the bean is constructed, the `loadData()` method is executed automatically due to the `@PostConstruct` annotation. In this application, the `loadData()` method deletes all data in the underlying datastore, populates it with two sample `User` records, returns all `User` records now stored in the database, and logs them to the console for verification.

NOTE: The Reactive Streams API and the implementation of it as provided by Spring WebFlux/Project Reactor is beyond the scope of this particular article. Please consult the appropriate documentation at the 'Web on Reactive Stack' Spring documentation site, any of several sessions I've delivered available on my YouTube channel, or by visiting the Reactive Streams and Project Reactor sites.

```
@Slf4j

@Component

@AllArgsConstructor

class DataLoader {

    private final UserRepository repo;



   @PostConstruct
   void loadData() {
       repo.deleteAll()
               .thenMany(Flux.just(new User("Alpha", "Bravo", "
                       new User("Charlie", "Delta", "1313 Mocki
```

```
                .flatMap(repo::save)
                .thenMany(repo.findAll())
                .subscribe(user -&gt; log.info(user.toString()))
    }


}
```

Demo

To access the environment variables set earlier as a result of sourcing the config script, we will need to build and run the Spring Boot application from the command line. From the project directory, execute the following command to build and run the project:

```
mvn spring-boot:run
```

To verify the application is able to access the Cosmos DB instance and return the correct data, execute the following command from another terminal window:

```
curl http://localhost:8080
```

Alternatively, you can access `http://localhost:8080` from a browser tab or window.

The following should be displayed:

```
» http :8080

HTTP/1.1 200 OK

Content-Type: application/json

transfer-encoding: chunked


[

    {

        "address": "123 N 4567th St",
```

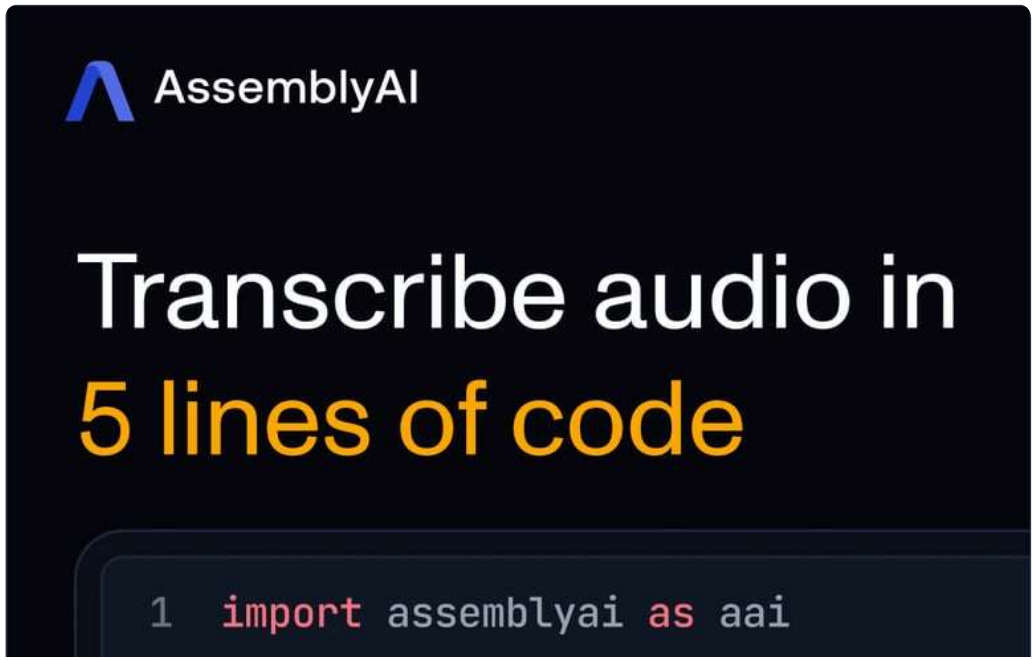
```
        "firstName": "Alpha",
        "id": "dcaca409-da66-4d53-9268-e6313b48bdd7",
        "lastName": "Bravo"
    },
    {
        "address": "1313 Mockingbird Lane",
        "firstName": "Charlie",
        "id": "14cf1b9b-f8c2-401b-affe-770fcaaacea3",
        "lastName": "Delta"
    }
]
```

Summary

One of the key benefits of using Spring Boot to develop Java (and Kotlin) applications is its developer-first focus. Spring Boot's flexibility and scalability, especially with regard to database connectivity, makes tapping into the power of a planetary-scale database such as Cosmos DB amazingly straightforward for developers. This lets devs concentrate on functionality that meets their stakeholders' needs and helps drive their organization forward, knowing that their data is safe, scalable, and accessible from anywhere Azure is reachable, e.g. planet Earth.

Resources

- How to use Azure Cosmos DB from your Spring Boot App (video) - beginning to 44:30
- Scripts to create Cosmos DB SQL database resources, instance
- Spring Boot project repository



Read More

Top comments (0)

Read the docs