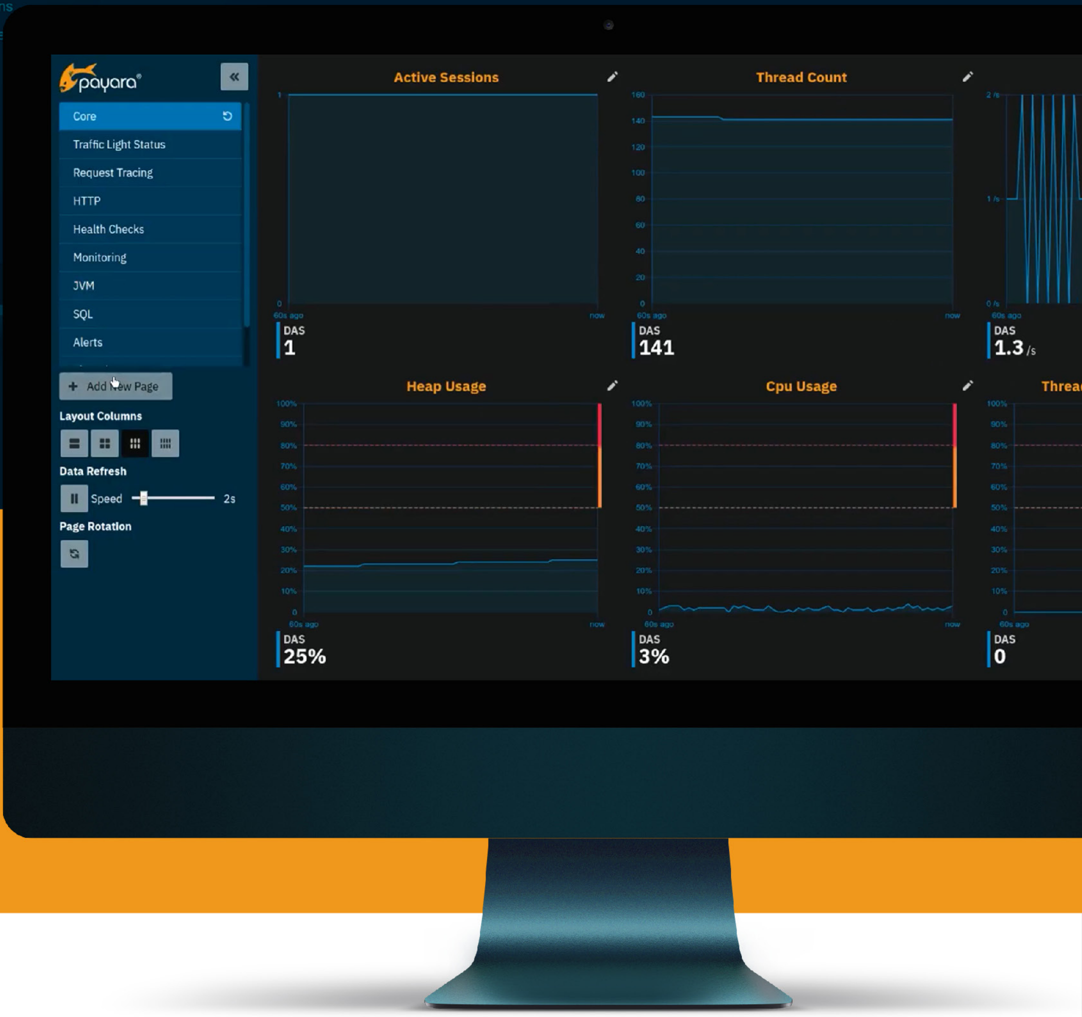
A Developer Guide to NoSQL Persistence on The Jakarta EE Platform With Google Firestore



The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

User Guide

Contents

Digital transformation, largely enabled by the commoditization of massive compute resources by cloud computing vendors has changed the way applications are developed, deployed and maintained. It has allowed for much faster application development iterations by organisations to meet ever changing and increasingly complex customer expectations.

The customised digital experiences pioneered by the likes of Amazon and Netflix has created a defacto standard of customer experience that all organisations aspire to in order to acquire and maintain customers. The COVID-19 pandemic also created a paradigm shift to more digital economic activities across the globe. Flowing from this transformation, is the vast amounts of data that are collected by various service providers. These collected data need to be stored in durable data stores.

The Java Platform has always had excellent support for relational database management systems (RDBMS) through the Java Database Connectivity (JDBC) API. The Jakarta EE Platform, hitherto known as Java EE, also has excellent support for SQL through abstractions such as the Jakarta Persistence API. With the rise in popularity of alternative data storage paradigms, enterprises have more choices outside of the traditional SQL data format for structuring, analaying, storing and querying application data.

This guide looks at using Jakarta EE with Firestore, the document NoSQL database from Google Firebase. We start by looking at the theory of Jakarta EE, then a look at the history, advantages and types of NoSQL databases, then how to set up Firestore in your Jakarta EE application, and then finally how to store and retrieve Jakarta EE application data. By the end of this guide, you will have a good grounding for practical exploration of Jakarta EE and NoSQL databases.

The Theory Of Jakarta EE

What Is Jakarta EE?

Jakarta EE is a set of community developed, abstract specifications that together form a platform for developing end-to-end, multi-tier enterprise applications. Jakarta EE is built on the Java Standard Edition, and aims to provide a stable, reliable and vendor neutral platform on which to develop cloud native applications.

Hitherto, Jakarta EE was called Java EE and was a property of Oracle Inc., evolved through the Java Community Process (JCP). However, in late 2017, Oracle decided to move the platform to an open foundation for a much broader community-led evolution. The Eclipse Foundation got chosen and Java EE, after the transfer, got rebranded to Jakarta EE.

What Is A Specification?

As stated in the above definition, Jakarta EE is made up of a set of specifications that each cover a specific API for solving a specific software development need. For example, the Jakarta Contexts and Dependency Injection (CDI) specification provides constructs for creating loosely coupled applications through dependency injection. These different specifications are combined into a single "umbrella" specification for each Jakarta EE release. As such, Jakarta EE 10 for instance, is released under the Jakarta EE 10 specification.

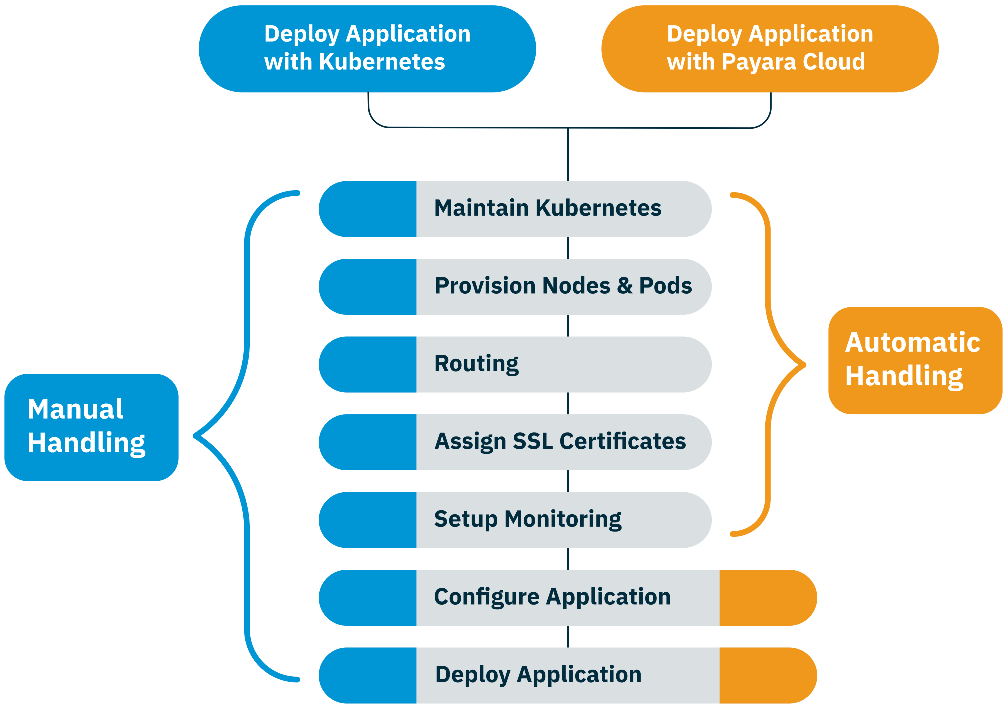
More technically, a specification is a formal proposal document made to the Jakarta EE Specification Committee through the Jakarta EE Specification Process (JESP) that outlines the functions of a given set of APIs. This document outlines what the expected behaviour should be for various invocations of the API. The specification then acts as the blueprint for the API.

What Is A Compatible Implementation?

As a specification is merely a document that outlines the behaviour of a given API, it needs an implementation that realises the actual outcomes for each invocation of the API. For instance, the Jakarta Persistence specification provides the EntityManager interface that has the persist() method. This method, when called and passed an instance of a Jakarta Persistence entity, persists that entity instance as a database row to the underlying database. The "library" that does the actual work of taking that instance and making sure it gets stored to the durable storage when the `EntityManager#persist()` method is invoked, is called a compatible implementation of the Jakarta Persistence specification.

Each specification that makes up the full Jakarta EE platform has an implementation. As a specification itself, the Jakarta EE platform also has an implementation in the form of compatible products. As the specifications are separated from their implementations, you as a developer will generally code against the API constructs of the specification, and are free to pick any compatible implementation of the platform. With this abstraction, Jakarta EE implementation vendors can collaborate on the base, standard specifications and compete through innovations on top of the base platform.

An example of such invocation is the Payara Cloud offering from Payara. This innovation helps you realise the dream of true separation of your business domain application and the runtime that powers it. With Payara Cloud, you simply upload your Jakarta EE application web archive (.war file) and have it automatically deployed to the cloud, just as Jakarta EE was envisaged to have separation of business domain from the runtime. Another example of custom features available on the Payara Platform is remote CDI events. This feature, built on the Jakarta CDI specification, allows the firing of CDI events that can be observed by any listener in a given Hazelcast cluster.



What Is Eclipse MicroProfile?

The Jakarta EE Platform is a general purpose platform for developing all kinds of applications. As modern application development paradigms have changed a lot in the past years, there is a need to evolve the platform to meet such changes. One such paradigm is cloud-native software application development.

As the base Jakarta EE Platform has always been geared towards enterprises, it has historically evolved at a much slower pace than changes in the software development space. It is for this reason that the Eclipse MicroProfile project was created as an extension to the base platform to provide cloud-native APIs for developing modern cloud-based applications.

Eclipse MicroProfile, built upon Jakarta CDI, Jakarta REST and Jakarta JSON Processing, comes with the following APIs

- OpenTracing
- OpenAPI
- REST Client
- Config
- Fault Tolerance
- Metrics
- JWT Propagation
- Health

These APIs augment the much larger Jakarta EE Platform APIs to provide the developer with a cohesive set of APIs for developing, testing and deploying cloud-native modern enterprise applications.

A Brief History Of NoSQL

Traditionally, enterprises stored their SQL data in relational database management systems (RDBMS). This system served and still serves its purpose very well. However, with the rise of the popularity of the aforementioned digital transformation in the early 2000s, coupled with the drop in storage prices, enterprises began to collect huge amounts of data that needed to be processed, analysed and stored, sometimes on the fly. Almost all of this massive amount of data was generated through web applications usage by people. The rapid change in delivering superior digital experiences to customers meant enterprises needed to change the way applications were developed.

One of the core changes had to do with how application data was structured. The much more strict and rigid SQL data format meant developers needed to model complex data ahead of time. However, the pace of change in delivering the new digital experiences meant that most of the time, application data had to go through many iterations during development. Enterprises began to coll

It was around this same time that the Agile Manifesto was gaining popularity. Agile meant faster application iteration, and this meant application data needed to be flexible to allow for much more fluidity. The rise of the commoditization of compute resources also meant that developers could spread their databases across multiple servers to scale out rather than scale up. Spreading application data across servers also meant the ability to place the data close to end users.

The Origin of The NoSQL Name

All these factors combined to popularise an already existing data storage paradigm name. In 1998, Carlo Strozzi, an Italian software engineer released a DBMS that did not use the SQL language for querying. He called his database NoSQL, to stand for a database that did not use SQL. His database however, was more in line with the existing relational model than the latter schemaless ones. Even though his database was relational, the name he had coined for it would come to be associated with a very opposite database technology.

The Rise of Schemaless Databases

The term NoSQL was again used by Johan Oskarsson and Eric Evans in 2009 to describe schema-less, non-relational databases. The distributed, non-relational model of NoSQL database technology made it the almost perfect choice for companies like Twitter, Google and Facebook that had amassed massive datasets from their web applications. These data sets needed to be analysed for insights and the much more flexible nature of the NoSQL database technology was a fit for purpose.

Big Data And NoSQL

The term Big Data was coined in 2005 by Roger Mougalas to refer to data that was mind bogglingly large such that it was impossible to process and manage using existing tools. This was the kind of data that the aforementioned companies were collecting towards the end of the first decade of this century. With the ability of NoSQL database systems to handle both structured and unstructured data, and their support for both for flexible horizontal scaling and on the fly data analysis, these companies started adopting the technology.

Naturally, NoSQL database technologies started gaining popularity. Different types we developed, culminating in different NoSQL database types. Almost all the current NoSQL databases are open source. The cloud vendors have developed their own NoSQL offerings as part of their cloud portfolio. As the cost of storage keeps going down, and enterprises collect large data sets, NoSQL database management systems play a pivotal role in storing and making sense of all this data.

Advantages of NoSQL Database Systems

NoSQL database systems, by their very nature, do have a number of advantages that can be appealing to enterprises coming from the relational world. Among these advantages are.

Flexible Data Models

The flexibility of NoSQL database systems means that the data can change without having to update the database. The data scheme can be considered "dynamic," changing to suit the data being inserted. This results in a lot of flexibility during development and bug fixing. It also means that development teams can iterate much faster, pushing out new features far faster than is possible when using RDBMS.

Faster Queries

NoSQL database systems are designed and optimised for querying. Unlike RDBMS that use JOIN for fetching data across multiple tables, NoSQL systems are optimised for queries through the way the data is stored. Most data that will be read together are stored together, rather than separately. This can result in much faster queries.

Scaling

NoSQL database systems allow for horizontal scaling-out, where you can add more commodity hardware when the need arises. This can result in significant cost savings compared to the typical vertical scale-up of RDBMS. With the rise in cloud vendors and massive commoditization of compute power, horizontal scaling can be a significant cost saving consideration.

Easier Developer Experience

The very flexible, schemaless nature of NoSQL means developers can turn out faster features. NoSQL data can also be mapped directly to the data models of the application programming language used by the developers. This allows for less context switches and reduces impedance mismatch.

Faster Application Iteration

The flexible nature of NoSQL means application development teams can iterate application features faster. The ability to have the database adjust to the application data model relieves development teams of having to model and map application data to the database. This in turn frees them to implement application features much faster.

Store Different Kinds Of Data

NoSQL systems allow the storage and retrieval of structured, semi-structured and unstructured data, depending on the need of the application. A typical enterprise application can have different types of data that need storing at different phases of application use. The flexible nature of NoSQL systems allow for the storing of all these kinds of data.

Types Of NoSQL

There are different types of NoSQL database systems, each geared towards a different application use case. Some databases might support multiple NoSQL types. One all the types have in common is the flexibility of the data schema.

Graph Databases

Graph NoSQL databases are designed for data whose relations can be represented or viewed as graphs. A typical example of graph data is social network relations. Each user has a graph relationship with their circles. These kinds of data are suited for graph databases. Examples of graph databases are Neo4J and JanusGraph.

Document Store

Document NoSQL databases store data as document objects, much like JSON objects. A typical document is an object that has key-value pairs, with the values being any supported database type like strings, numbers, and even other objects. A database in document NoSQL data parlance is called a collection, and within collections, you have documents, analogous to tables in an RDBMS. However, unlike RDBMS tables, the structure of documents in a collection can differ from each other thanks to its flexible nature. Examples of document databases are Google Firestore and Mongodb.

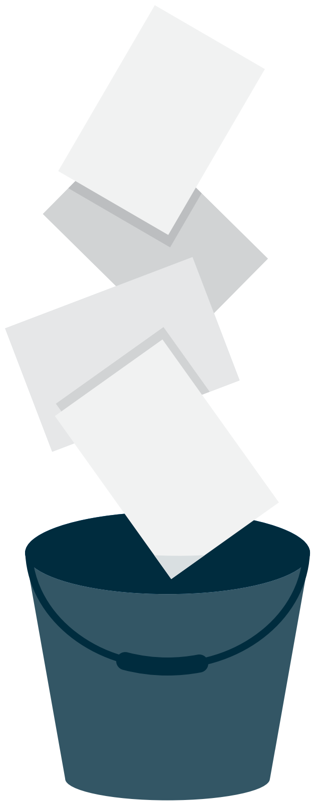
Key-Value

Key-value databases store their data as dictionaries, with each entry having a key and associated value. Within the dataset, each key can be used once, much similar to how a map works in Java. Key-value databases are very popular as caches for storing intermediate application data in-memory. Given their very nature, key-value databases can be incredibly fast for data retrieval. Examples include Redis and Couchbase.

Wide-Column

A wide-column database stores its data into flexible columns that can span multiple servers in a cluster. The names and format of columns can be different across different rows. By their very nature, queries in a column are very fast. Examples of wide-column databases are Apache Cassandra and Google BigTable.

Jakarta EE And Google Firestore



Google Firestore is a fully hosted NoSQL database offering from Google's Firebase service. Firestore is schemaless and document oriented. You store your data in documents that are collected into collections. A collection is a container for holding documents. A collection in Firestore is just a bucket. It cannot have any attribute whatsoever.

Set Up

As a Java library, you can integrate the Firebase API into your Jakarta EE application using some constructs like CDI producers. First, let us pull in the dependency.

The Firebase Maven Dependency

Setting up Firestore in your Jakarta EE application entails adding the firebase-admin dependency to your project as shown below.

```
<dependency>
  <groupId>com.google.firebase</groupId>
  <artifactId>firebase-admin</artifactId>
  <version>9.1.1</version>
</dependency>
```

This dependency pulls in the necessary APIs to be able to connect to Google Firebase to use the Firestore database.

Producing The Firestore Object

The root entry to the Firestore database service is the com.google.cloud.firestore.Firestore interface. This interface contains all the methods for interacting with the service. As Jakarta EE allows us to organise an application with CDI, rather than litter the code instantiating this object everywhere, we can have a central “factory” for producing it as shown below.

```
@ApplicationScoped
@Log
public class DocFactory {

      @Produces
      @ApplicationScoped
      public Firestore initFirestoreDb() {

            try {
                  final InputStream resourceAsStream = this.getClass().
getResourceAsStream("/service-account-file.json");
                  assert resourceAsStream != null;
                  final FirebaseOptions firebaseOptions =
                              FirebaseOptions.builder()
                                          .setCredentials(GoogleCredentials.
fromStream(resourceAsStream))
                                          .build();
                  FirebaseApp.initializeApp(firebaseOptions);
                  log.log(Level.INFO, () -> "Returning a FirestoreClient to
class --> " + injectionPoint.getBean().getName());
                  return FirestoreClient.getFirestore();

            } catch (Exception e) {
                  throw new RuntimeException(e);
            }

      }
}
```

The DocFactory is a CDI ApplicationScoped singleton that has a producer method, initFirestoreDb that "produces" an ApplicationScoped Firestore object. The method uses the file instantiation method for authenticating to the Firebase service. There are a number of ways of authenticating to the service, one of which is downloading a JSON file from the service and using it to authenticate by passing it to the `com.google.auth.oauth2.GoogleCredentials` utility class. In a production app, the JSON auth file should be loaded from somewhere on the host machine. It should not be bundled with the application.

With the Firestore producer in place, using it is just a matter of doing `@Inject` into an injection point in a bean. As the returned Firestore instance from the producer method in DocFactory is an application scoped singleton, we benefit from not having to instantiate it over and over anytime an instance is needed.

CRUD With Firestore

Almost every single application has, at its core, the creation, reading, updating and sometimes destruction of data. We can equally do the same with Jakarta EE and Firestore. Let us start with creating data.

Create

Creating or storing data in Firestore from your Jakarta EE application is a very straightforward process. However, it is important to not get carried away by the schemaless, flexible nature of the datastore and end up not structuring your data models well. The root cause of many application scaling problems can be traced to how the application data models are constructed. As such, it is important to understand how the datastore sees and stores information and then structure your models within that context, taking your business domain into account.

The sample data model for this guide is made up of two classes, a Department and Employee. A department can have many employees, and an employee can have at most one department. This is a quintessential paradigm in a typical Jakarta EE application. The Department.java class is shown below.

```
public class Department {
      private String id;
      private String departmentName;
      private String description;
      private String businessKey;
      private List<String> employeeBusinessKeys = new ArrayList<>();

      private LocalDateTime createdOn;
      private LocalDateTime updatedOn;

      public void stamp() {
            if (createdOn == null) {
                  setCreatedOn(LocalDateTime.now(ZoneOffset.UTC));
            }
            if (id != null && !id.isBlank()) {
                  setUpdatedOn(LocalDateTime.now(ZoneOffset.UTC));
            }
      }

}
```

This is a simple Plain Old Java Object. The String list field contains the business key of all employees in a given department. The `stamp()` method sets some basic lifecycle timestamps. The `Employee.java` class is shown next.

```
public class Employee {

      private String id;

      private LocalDateTime createdOn;
      private LocalDateTime updatedOn;

      @NotBlank
      private String firstName;
      private String middleName;
      @NotBlank
      private String lastName;
      private String businessKey;
      @NotNull
      private LocalDate dateOfBirth;
```

```
    private String departmentBusinessKey;

    public void stamp() {
        if (createdOn == null) {
            setCreatedOn(LocalDateTime.now(ZoneOffset.UTC));
        }
        if (id != null && !id.isBlank()) {
            setUpdatedOn(LocalDateTime.now(ZoneOffset.UTC));
        }
    }

}
```

The Employee class is equally a simple POJO with a link back to the Department in which an employee is. It also has the utility `stamp()` method. Both classes have business key fields. This concept warrants some explanation.

Business Vs Technical IDs

Every data stored in a datastore must have a unique identifier to identify it within the datastore. In a RDBMS, this is the primary key. In a NoSQL datastore, this could be the document ID or some other construct peculiar to the datastore. One thing that is common however, is that these IDs are more technical than business. They are technical in the sense that they have a predefined purpose, that is to identify each data entry in the datastore uniquely. The id fields in both the Department and Employee classes refer to the document reference of each instance that will be inserted into the collection.

Some applications use these technical IDs in the application for identifying data. This is not wrong. However, from a business perspective, it is good practice to assign unique business context identifiers to data as well. So a given unit of data to be stored in a data store will have a technical ID, almost always automatically set or generated by the datastore, and a business ID that should be set by the application. This business ID should have some form of meaning within the context of the application and business domain.

For instance the business ID of an insurance policy could be the policy number. The business ID of a department could be a combination of the department name and some meaningful serial number. The business ID of an employee could be a combination of the employee department and some employee specific serial number. The concept of business ID is very business specific and as such will need to be decided based on the context.

For the Department and Employee classes shown above, both are linked to each other through their business IDs. For brevity and simplicity, this guide uses random UUID as the business ID for both. However in your application, you should have a dedicated controller for generating such IDs.

Saving A Department

The most atomic unit of data in Firestore is the document. A document is much similar to JSON, with specific supported types. A document can be roughly equated to a database row in a RDBMS. Documents are collected together in a collection. The collection's single role is to act as a receptacle of documents and nothing else. A document can have its ID set by Firestore or you set it manually. My recommendation is to let the database set it. A collection can have any name.

A document is schemaless, meaning one entry can have completely different fields from the next one. This gives you a lot of flexibility in designing your data model. However, with such flexibility comes the complexity of ensuring your data models are properly designed. You should also think about how to validate your data to ensure data consistency and integrity in your application. The code below shows how to add data to a collection. In this case, we add a Department instance to a collection called jakarta-ee.

```
@ApplicationScoped
public class PersistenceService {
      public static final String BUSINESS_KEY_FIELD = "businessKey";
      public static final String JAKARTA_EE_COLLECTION = "jakarta-ee";
      @Inject
      Firestore db;

      Type type;
      DocumentReference documentReference;
      CollectionReference collectionReference;

      @PostConstruct
      void init() {
            collectionReference = db.collection(JAKARTA_EE_COLLECTION);
            type = new TypeLiteral<Map<String, Object>>() {
            }.getType();

      }



      public Department saveDepartment(final Department department) {
            documentReference = collectionReference.document();

            if (isEmptyString(department.getBusinessKey())) {
                  department.setBusinessKey(UUID.randomUUID().toString());
            }
            department.stamp();
            department.setId(documentReference.getId());

            documentReference.set(toJsonMap(department));
            return findDepartment(department.getBusinessKey());
      }}
```

The PersistenceService is an ApplicationScoped singleton bean that manages interactions with the database. We instantiate a `com.google.cloud.firestore.CollectionReference` object in the `@PostConstruct` method of the bean by calling collection method on the injected Firestore database instance, passing in the name of the collection, in our example, jakarta-ee. If the collection does not exist, it will be created automatically.

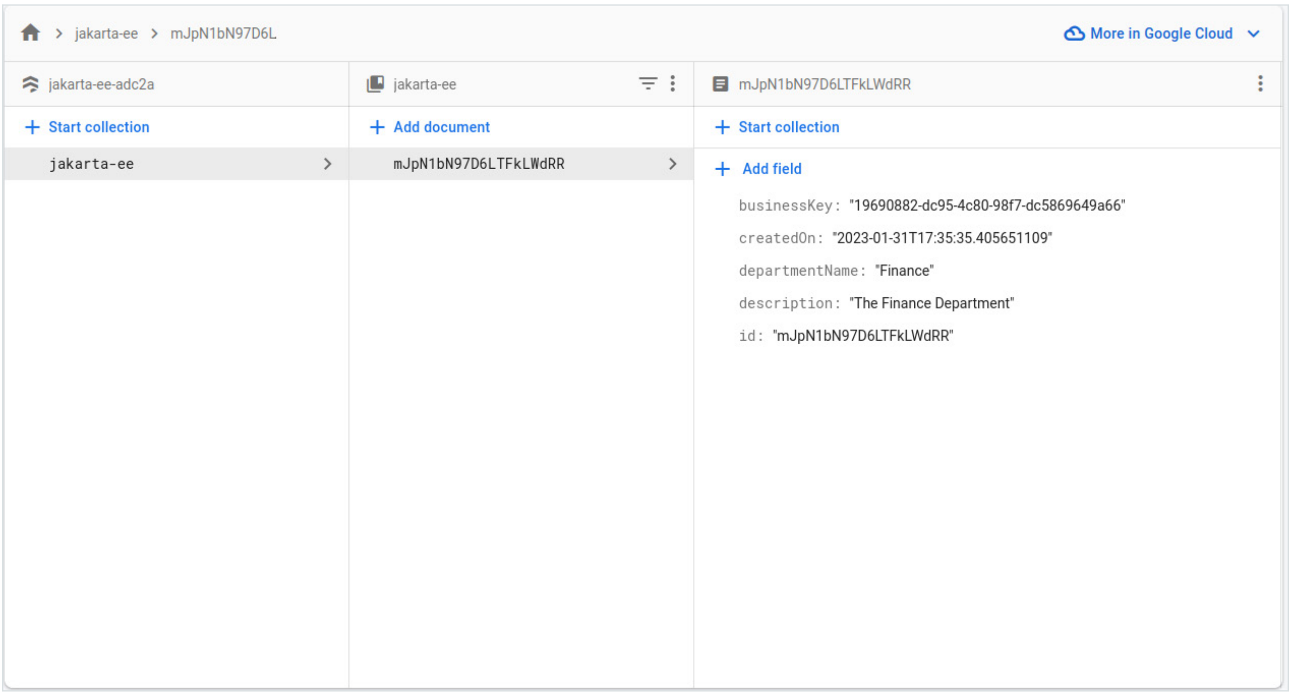
The saveDepartment method first calls the document method on the CollectionReference instance, without passing in any argument. This returns a `com.google.cloud.firestore.DocumentReference` instance with an auto generated document reference ID. The method sets a business ID if there is none set. The returned DocumentReference ID is set on the department instance in the id field. This way, each instance has a technical ID of the document into which it is stored.

The set method on the DocumentReference is what actually passes the data to the database. This method has overridden versions that take either a POJO or a `java.util.Map<String, Object>`. The Firestore database instance will automatically convert passed Java objects to the underlying database JSON document representation. However, directly passing in Java objects to the set method on the DocumentReference could limit your models to only types that the underlying marshaller supports. For example you cannot freely marshal or unmarshal a number of the java.time types using the POJO feature of the Firestore database.

To workaround that, we convert the Department instance to a JSON representation of `Map<String, Object>` and pass this converted String to the set method. This way, we can use the gamut of Java types (including `java.time` types) and still be able to persist data to the Firestore database. The toJsonMap method is shown below.

```
private Map<String, Object> toJsonMap(final Object object) {
            return json.fromJson(json.toJson(object), new
TypeLiteral<Map<String, Object>>() {
            }.getType()
);
      }
```

This utility method uses the `jakarta.json.bind.Jsonb` instance to convert an object to a `Map<String, Object>` representation. Such a map will have each field as a key and the field value as the value of the key. Calling the set method on the DocumentReference instance will persist the passed instance to a document in the collection from which the DocumentReference was obtained. An example of such a call will result in data in the jakarta-ee collection as shown below:



```
businessKey: "19690882-dc95-4c80-98f7-dc5869649a66"
createdOn: "2023-01-31T17:35:35.405651109"
departmentName: "Finance"
description: "The Finance Department"
id: "mJpN1bN97D6LTFkLWdRR"
```

Saving An Employee

The process of saving an Employee instance into the Firestore database is not much different from how the Department was saved above, as shown in the following code snippet.

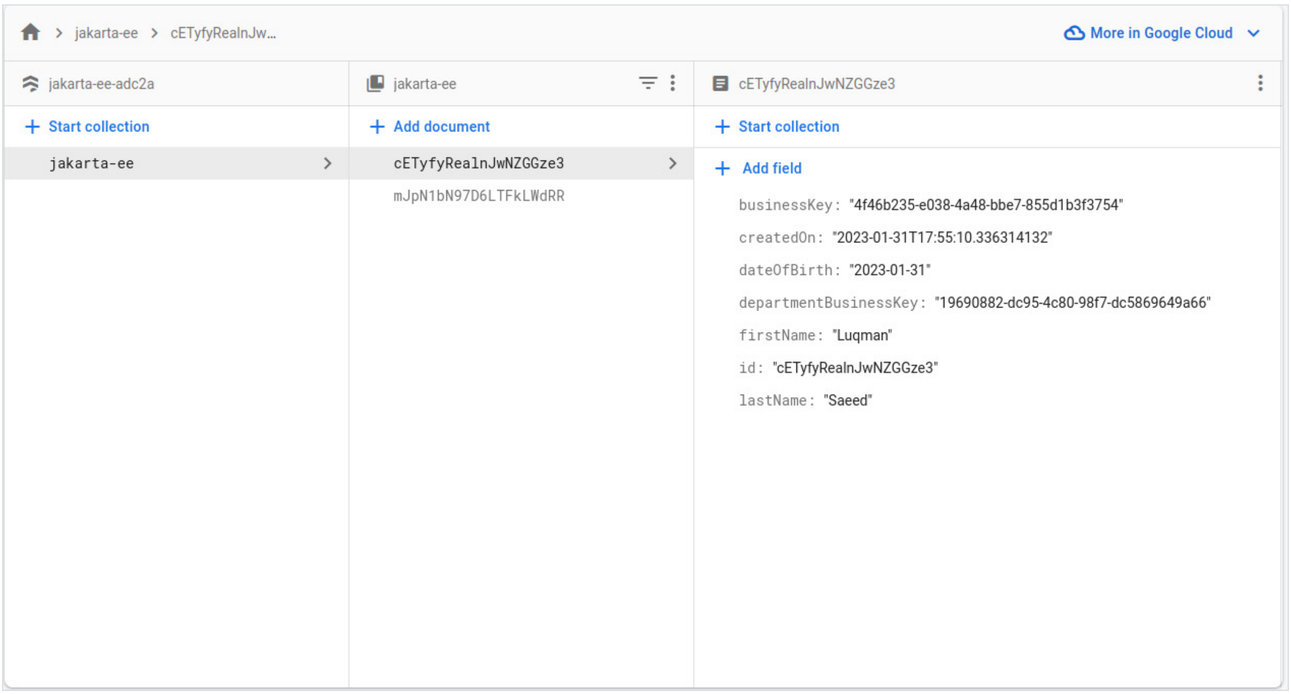
```
public Employee saveEmployee(final String department, final Employee employee)
{
            documentReference = collectionReference.document();

            Department dept = findDepartment(department);
            if (dept != null) {
                  documentReference = collectionReference.document();

                  if (isEmptyString(employee.getBusinessKey())) {
                        employee.setBusinessKey(UUID.randomUUID().toString());
                  }
                  employee.stamp();
                  employee.setId(documentReference.getId());
                  employee.setDepartmentBusinessKey(dept.getBusinessKey());
                  if (dept.getEmployeeBusinessKeys() == null) {
                        dept.setEmployeeBusinessKeys(new HashSet<>());
                  }
                  dept.getEmployeeBusinessKeys().add(employee.getBusinessKey());
                  documentReference.set(toJsonMap(employee));

                  updateDepartment(dept);
                  return findEmployee(employee.getBusinessKey());
            }
            throw new RuntimeException("No department found for dept business
key " + department);
      }
```

The saveEmployee method takes the business key of a department to which the new employee should be assigned. The Department is loaded from the database, the employee business ID is set and then the employee gets added to the loaded department, and has its department linkback also set. The employee instance is then persisted, and the loaded department is also then updated. A sample call to this method results in an employee instance in the database as shown below.

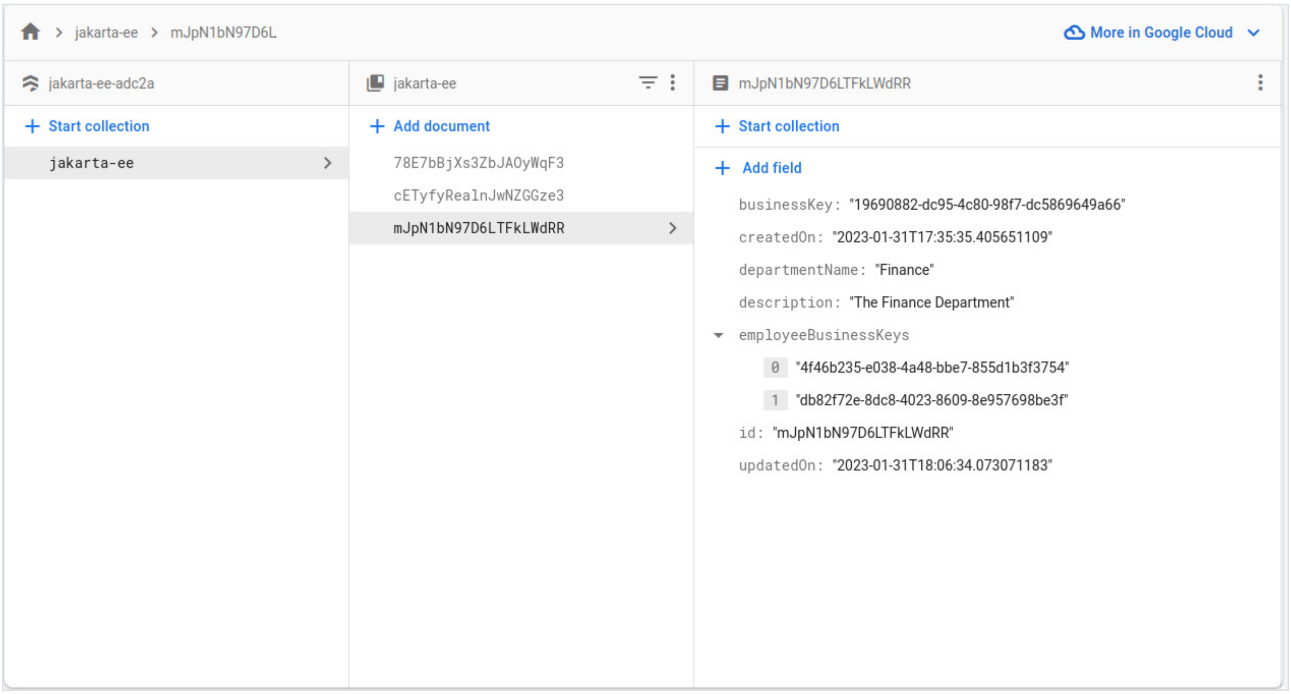


jakarta-ee
cETyfyRealnJwNZGGze3
mJpN1bN97D6LTFkLWdRR

cETyfyRealnJwNZGGze3

businessKey: "4f46b235-e038-4a48-bbe7-855d1b3f3754"
createdOn: "2023-01-31T17:55:10.336314132"
dateOfBirth: "2023-01-31"
departmentBusinessKey: "19690882-dc95-4c80-98f7-dc5869649a66"
firstName: "Luqman"
id: "cETyfyRealnJwNZGGze3"
lastName: "Saeed"

The employee instance has a linkback to its department through the `departmentBusinessKey` field. The updated department in the database is also shown below.



```
businessKey: "19690882-dc95-4c80-98f7-dc5869649a66"
createdOn: "2023-01-31T17:35:35.405651109"
departmentName: "Finance"
description: "The Finance Department"
employeeBusinessKeys
    0 "4f46b235-e038-4a48-bbe7-855d1b3f3754"
    1 "db82f72e-8dc8-4023-8609-8e957698be3f"
id: "mJpN1bN97D6LTFkLWdRR"
updatedOn: "2023-01-31T18:06:34.073071183"
```

Creating documents in Firestore entails calling the set method on a DocumentReference. As we have seen, a neat workaround to avoiding pitfalls is converting a POJO to a JSON map, passing such a map to the set method. This I have found to be the most flexible way to maintain the gamut of Java features and still be able to take full advantage of Firestore.

Reading Data

Reading data from the Firestore database is not much different from how it is done in a RDBMS. You can either directly load a document by its document ID or reference, or query.

Fetching By Document Reference

Fetching a document by its reference entails passing the document reference or technical ID to the document method on a `com.google.cloud.firestore.CollectionReference` instance. For instance, to fetch a Department by its reference, or primary key if you will, we make the following call.

```
Map<String, Object> dept =
collectionReference.document("mJpN1bN97D6LTFkLWdRR").get().get().getData();
```

The returned Map<String, Object> can then be transformed into the concrete Java type with the following code snippet.

Department department = json.fromJson(json.toJson(d.getData(), Department.class);

This mode of fetching data works fine as long as you don't mind the mix of technical and business ID in your code base.

Fetching Data By Querying

The Firestore database has rich query capabilities. The CollectionReference class extends com.google.cloud.firestore.Query class to give it powerful methods for querying your data. The following code snippet shows querying for a Department by its business key.

```
public Department findDepartment(final String businessKey) {
           ApiFuture<QuerySnapshot> query =
                      collectionReference.whereEqualTo(BUSINESS_KEY_FIELD,
businessKey).get();
           try {
                 List<QueryDocumentSnapshot> documents = query.get().
getDocuments();
                 Department department = documents.stream().map(d -> json.
fromJson(json.toJson(d.getData()), clazz)).findFirst().orElse(null);
                 log.log(Level.INFO,
                             () -> String.format("Returning department %s with
business key %s", json.toJson(department),
                                         businessKey));
                 return department;
           } catch (final Exception e) {
                 throw new RuntimeException(e);
           }
     }
```

The `CollectionReference#whereEqualTo` method takes a field in the document and a value to compare to. It returns a `com.google.cloud.firestore.QuerySnapshot` wrapped in a `com.google.api.core.ApiFuture`. The `getDocuments` call on the returned QuerySnapshot returns a list of `com.google.cloud.firestore.QueryDocumentSnapshot`. This is then streamed and mapped to a Department Java type.

Firebase querying is very expressive and has a lot of options that give you the flexibility to query your data in from different angles. You can do all the traditional paging, sorting, filtering that you may have been used to doing with RDBMS. Do check out the docs for the various querying options and features.

Updating Data

Updating a given document in Firestore is done by passing a com.google.cloud.firestore.SetOptions to the set method of DocumentReference. The updateDepartment method below shows how to update a Department, much similar to how it is done in a RDBMS.

```
public Department updateDepartment(final Department department) {
            DocumentReference document = collectionReference.
document(department.getId());
            department.stamp();
            document.set(toJsonMap(department), SetOptions.merge());
            return findDepartment(department.getBusinessKey());
      }
```

Much like persisting data, updating a given document requires getting a reference to the document, in the above case, using the id field in a passed instance. Since we set the id field to the technical id of the document, we can use that to get a reference to the document from Firestore. With the reference in hand, we pass in the updated department as a JSON map, passing in a second option of `SetOptions.merge()` to the set method on the document reference.

This call will cause Firestore to update the document in the database with the newly passed instance. Only fields with values in the newly passed instance will be updated in the database. The setOptions has other methods to control the update process. If you are coming from a RDBMS, then the SetOptions.merge() method as used above will be familiar.

Deleting Data

To delete a document in Firestore, call the `delete()` method on a given DocumentReference. This will remove the Document from the collection. As a precaution, I don't encourage the act of deleting data except for legal purposes. I recommend you archive or anonymise data rather than an outright deletion, again except for legal reasons.

Caveats

As a Jakarta EE developer, you most likely are familiar with the Jakarta Persistence API that allows you to harness the full power of RDBMS in your Jakarta EE applications. Firestore is a completely different data paradigm as showcased so far in this guide. There are some caveats you should be aware of as you take this NoSQL database for a ride.

No Cascade Operation

Unlike in Jakarta Persistence where you can model entities with relationships that cascade operations, Firestore has no such feature. If you need it, you might have to roll it out by hand as we did with persisting an Employee and then having to manually update the corresponding Department.

Database Referential Integrity

The referential integrity feature of a typical RDBMS is not available in the Firestore database. If you heavily rely on that feature, you might want to keep this in mind and find a workaround.

Different Query Paradigm

NoSQL queries are different from traditional SQL queries. Most notable is the absence of table JOINs. This is made up for in most NoSQL data modelling by data duplication to allow for “single shot” queries. NoSQL queries in general are fundamentally different from SQL counterparts because the underlying data is different. Keep this in mind when giving NoSQL a trial.

Different Data Model

NoSQL databases have a different data modelling paradigm. This is important to keep in mind so that you can design your data models in a way that allows you to take maximum advantage of NoSQL features. Ideally you should not approach NoSQL data modelling with the RDBMS mindset. NoSQL databases like Firestore offer much flexibility at the cost of some data duplication. This is something to keep in mind when designing your models. You might have to duplicate some data to optimise your queries.

Summary

This guide has introduced you to incorporating Firestore NoSQL database into your Jakarta EE application. In the process, you learned the theory of Jakarta EE, a brief history of NoSQL, types of NoSQL databases and finally incorporated Firestore database into your application and saw how to make basic CRUD operations using the Firestore database. You should now have a firm foundation on which you can explore the use of NoSQL in your own projects.

You may find these guides useful:

- A Business Guide to NoSQL on the Jakarta EE Platform
- The Complete Guide to Testing on the Jakarta EE Platform
- The Complete Guide To JSON Processing On the Jakarta EE Platform