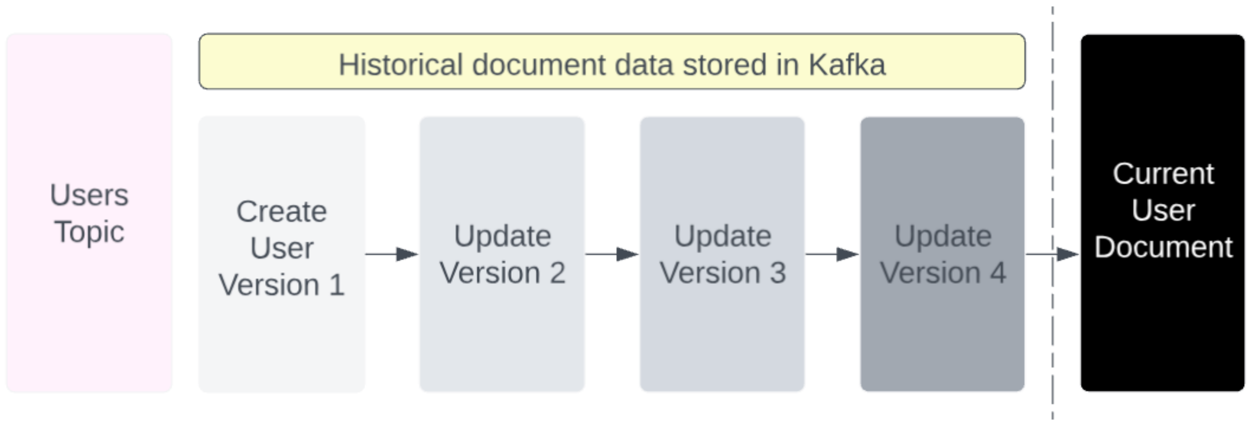
NucleoDB

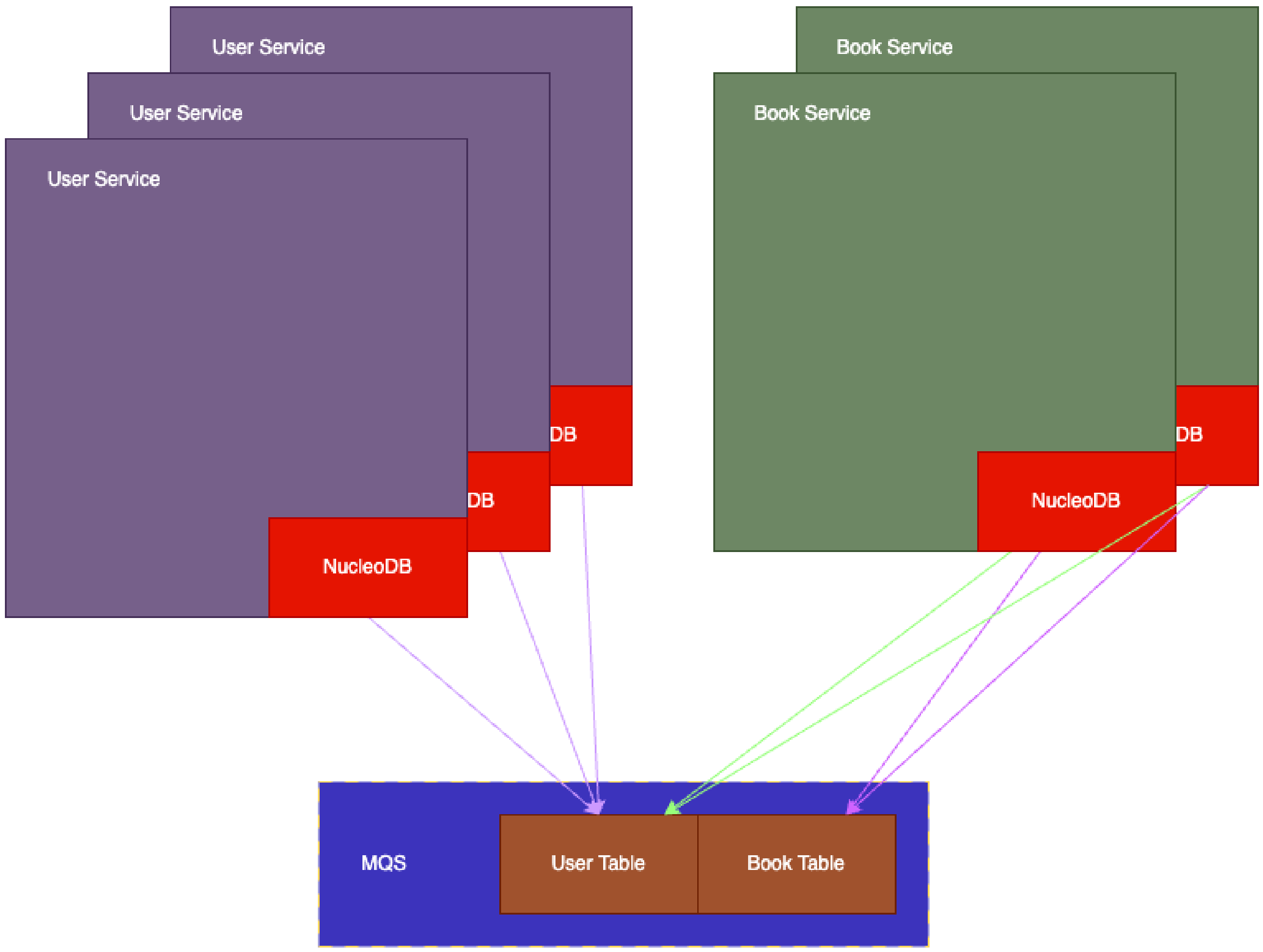
Description



NucleoDB is an in-memory, embedded database system designed to provide high-speed data management and processing. Its in-memory architecture ensures rapid access and manipulation of data, making it ideal for applications where performance and quick response times are critical. Being embedded, NucleoDB seamlessly integrates into various applications, offering a streamlined and efficient way to handle data within the application's own environment. This design approach not only simplifies the application architecture but also enhances overall system performance by reducing the need for external data calls.

One of the key features of NucleoDB is its interaction with Kafka, a popular distributed streaming platform. NucleoDB connects directly to a Kafka cluster, with each table within NucleoDB corresponding to a separate Kafka topic. This setup facilitates efficient data streaming and synchronization between NucleoDB and Kafka, enabling real-time data processing and analytics. Additionally, NucleoDB includes a specialized 'Connections' table, reserved for linking two entries from different tables together, along with associated metadata. This feature allows for the creation of complex relationships and data structures within the database, enhancing its capability to handle diverse and intricate data models. The integration with Kafka, combined with its in-memory and embedded qualities, positions NucleoDB as a powerful tool for modern, data-driven applications.

Architecture


User Service
User Service
User Service
DB
DB
NucleoDB
Book Service
Book Service
DB
NucleoDB
MQS
User Table
Book Table

Requirements

At the moment the database only has compatibility with Kafka.

Kafka Cluster

You can run a development cluster using docker and docker-compose using the following file.

https://github.com/NucleoTeam/NucleoDB-Spring/blob/main/docker/kafka/docker-compose.yml

```
version: "2"

networks:
  nucleodb-network:
    driver: bridge
services:
  zookeeper:
    image: docker.io/bitnami/zookeeper:3.7
    ports:
      - "2181:2181"
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
    networks:
      - nucleodb-network
  kafka1:
    image: bitnami/kafka:latest
    ports:
      - 29092:29092
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_BROKER_ID=1
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,EXTERNAL://0.0.0.0:29092
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka1:9092,EXTERNAL://127.0.0.1:29092
      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,EXTERNAL:PLAINTEXT
    depends_on:
      - zookeeper
    networks:
      - nucleodb-network
  kafka2:
    image: bitnami/kafka:latest
    ports:
      - 29093:29093
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_BROKER_ID=2
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9093,EXTERNAL://0.0.0.0:29093
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka2:9093,EXTERNAL://127.0.0.1:29093
      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,EXTERNAL:PLAINTEXT
    depends_on:
      - zookeeper
    networks:
      - nucleodb-network
  kafka3:
    image: bitnami/kafka:latest
    ports:
      - 29094:29094
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_BROKER_ID=3
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9094,EXTERNAL://0.0.0.0:29094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka3:9094,EXTERNAL://127.0.0.1:29094
      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,EXTERNAL:PLAINTEXT
    depends_on:
      - zookeeper
    networks:
      - nucleodb-network
```

Quickstart

Add dependencies to build.gradle

```
repositories {
  mavenCentral()
  maven { url 'https://repo.spring.io/milestone' }
  maven { url 'https://repo.spring.io/snapshot' }
  maven { url "https://nexus.synload.com/repository/maven-repo-releases/" }
}
dependencies {
  implementation 'com.nucleodb:library:1.15.11'
}
```

Create models

Models consist of 2 classes, the DataEntry class that wraps the data class.

Data Class

```
import com.nucleodb.library.database.tables.annotation.Table;
import java.io.Serializable;

@Table(tableName= "book", dataEntryClass = BookDE.class)
public class Book implements Serializable{
  private static final long serialVersionUID = 1;
  @Index String name;
  public Book(String name){
    this.name = name;
  }
  ....getters/setters
}
```

Data Entry Class

```
import com.nucleodb.library.database.index.annotation.Index;
import com.nucleodb.library.database.tables.table.DataEntry;
import com.nucleodb.library.database.modifications.Create;

public class BookDE extends DataEntry<Book>{
  private static final long serialVersionUID = 1;
  public BookDE(Book obj) {
    super(obj);
  }

  public BookDE(Create create) throws ClassNotFoundException, JsonProcessingException {
    super(create);
  }

  public BookDE() {
  }

  public BookDE(String key) {
    super(key);
  }
}
```

Instantiate the database

```
NucleoDB nucleoDB = new NucleoDB(
    NucleoDB.DBType.NO_LOCAL, // no local cache
    "com.package.location"
);
DataTable table = nucleoDB.getTable(Book.class); // get the table by the table class
```

Database usage

Looking up by index using get({key}, {value})

```
// saving/inserting
BookDE book = new BookDE(new Book("The Grapes of Wrath"));
table.saveSync(book);

// read only access
Set<DataEntry> entries = table.get("name", "The Grapes of Wrath");

// write access
Set<DataEntry> entries = table.get("name", "The Grapes of Wrath", new DataEntryProjection(){{
  setWritable(true);
  setLockUntilWrite(true); // cluster wide lock for entry until save (or 1 second timeout)
}});

// delete
BookDE book = (BookDE) entry;
table.deleteSync(book.copy(BookDE.class, true));
```

Models

Table Class

```
@Table(tableName = "author", dataEntryClass = AuthorDE.class)
public class Author implements Serializable {
  private static final long serialVersionUID = 1;

  @Index()
  String name;

  public Author() {}

  public Author(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

@Table

```
@Table(tableName = "author", dataEntryClass = AuthorDE.class)
```

tableName indicates the name of the table but also the topic that will be used by the MQS.

dataEntryClass points to the data entry class that encloses this table data class.

serialVersionUID

For the local cache this is needed for serializing and deserializing the data entries in the database. This will speed up startup for subsequent startups and will not grab all entries from the MQS.

@Index

This member variable will be indexed with the name of the variable. Nested objects in the data table class can also be indexed. You can specify the indexed key name by giving the Index annotation a value.

```
@Index("keyName")
```

When using a custom index key name you will need to use this key name for lookups.

DataEntry Class

Data Entry class encloses the table class. This is to differentiate the meta data from the actual data of the table. *Only the table class can have indexed member variables.*

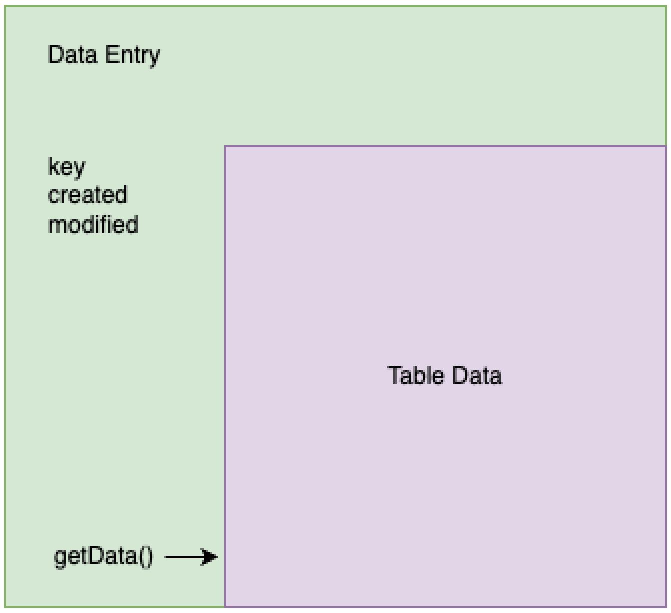
Author class

```
public class AuthorDE extends DataEntry<Author>{
  public AuthorDE(Author obj) {
    super(obj);
  }

  public AuthorDE(Create create) throws ClassNotFoundException, JsonProcessingException {
    super(create);
  }

  public AuthorDE() {
  }

  public AuthorDE(String key) {
    super(key);
  }
}
```

DataEntry

```
public class AuthorDE extends DataEntry<Author>
```

DataEntry contains metadata definitions and in the case above the *Author* class contains table data definitions. Metadata is maintained by the NucleoDB database. Modified/Created dates are based on the ledger in the MQS.



Constructors

Constructors are required for internal operations and instantiating a new DataEntry into the database.

Below is used when inserting a new Author table class into the database. This will also generate a new key using UUID.randomUUID().

```
public AuthorDE(Author obj) {
  super(obj);
}
```

Below is used when a Create is read in from the MQS

```
public AuthorDE(Create create) throws ClassNotFoundException, JsonProcessingException {
  super(create);
}
```

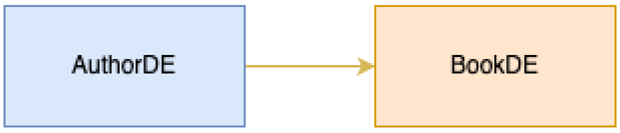
Connection

```
@Conn("AUTHORED")
public class AuthoredConnection extends Connection<BookDE, AuthorDE>{
  public AuthoredConnection() {
  }

  public AuthoredConnection(AuthorDE from, BookDE to) {
    super(from, to);
  }

  public AuthoredConnection(AuthorDE from, BookDE to, Map<String, String> metadata) {
    super(from, to, metadata);
  }
}
```

Connection

```
public class AuthoredConnection extends Connection<BookDE, AuthorDE>{
```

In the example above the AuthorDE is pointing to BookDE in a OneToMany relationship



@Conn(name)

Configures the name of the connection and the topic name used in MQS.

Indexing

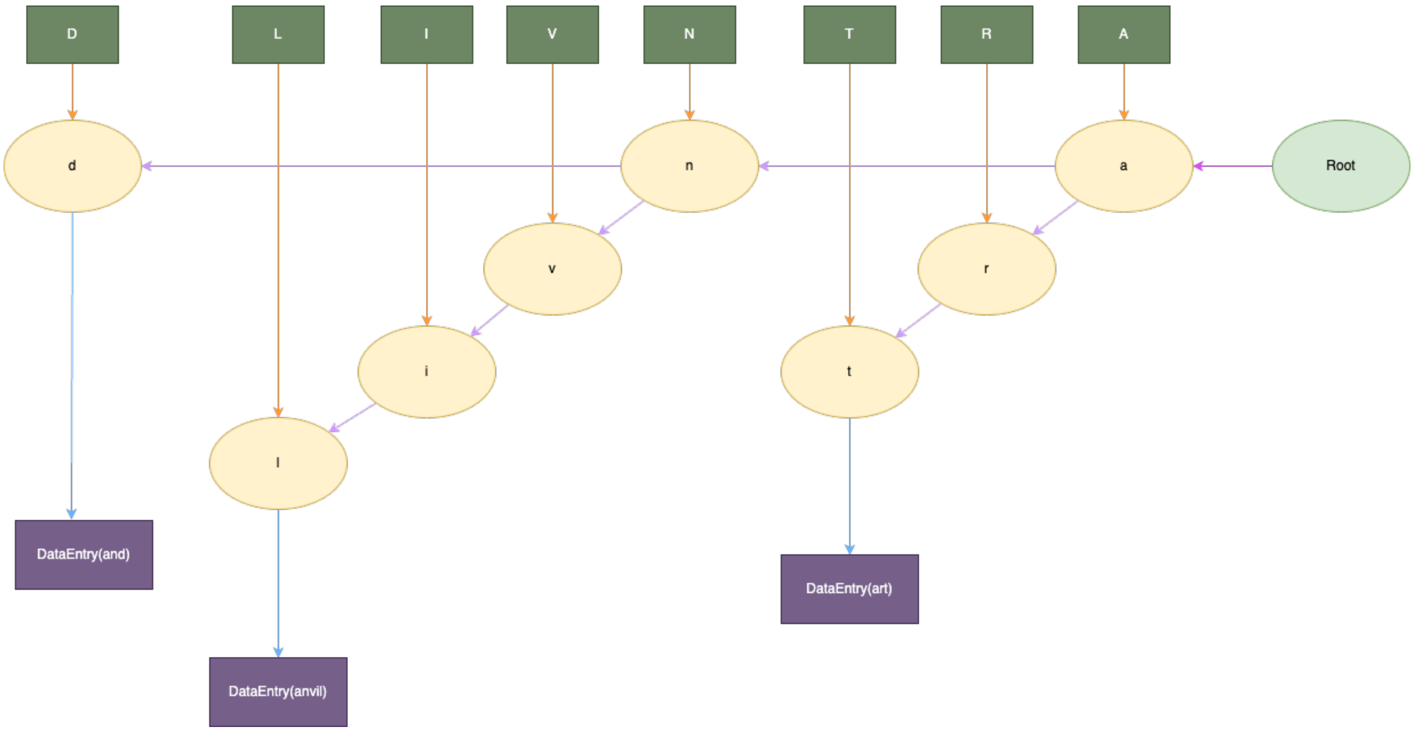
Currently there are 2 index types, **Tree** and **Trie**.

Trie Index

Supported value types: String

Heavy memory usage and allows for partial text search.

```
@Index(type = TrieIndex.class)
```



Tree Index (Default)

Support value types: Numbers, String, Dates

Lower memory usage and allows for lookup by object.

```
@Index(type = TreeIndex.class)
```

MQS

Message Queue Service

Setting MQS

In the configuration for the table you can set the MQS type on initiating the database.

```
config.setMqsConfiguration(new LocalConfiguration())
```

This can be done using the configuration customizer

```
NucleoDB nucleoDB = new NucleoDB(
    NucleoDB.DBType.READ_ONLY,
    c -> c.getConnectionConfig().setMqsConfiguration(new LocalConfiguration()),
    c -> c.getDataTableConfig().setMqsConfiguration(new LocalConfiguration()),
    "com.nucleodb.library.helpers.models"
);
```

You can specify specific tables by comparing the table class

```
c -> {
  if(c.getClazz() == Author.class){
    c.getDataTableConfig().setMqsConfiguration(new LocalConfiguration());
  }
},
```

Kafka MQS (Default)

Using Kafka cluster to keep the ledger of the database.

Configuration Path

```
com.nucleodb.library.mqs.kafka.KafkaConfiguration
```

Configure

This MQS is configured using environment variables.

```
KAFKA_SERVERS=127.0.0.1:19092,127.0.0.1:29092,127.0.0.1:39092
KAFKA_GROUP_ID=single-node-db
```

Or in the settings map

```
servers=127.0.0.1:19092,127.0.0.1:29092,127.0.0.1:39092
groupName=498yrhebfnsfhsdjfad
```

Settings map can be configured in the configuration customizer.

Local MQS

Local only for testing purposes. Does not use a message queue, all messages are instantly produced directly to the consumer.

Configuration Path

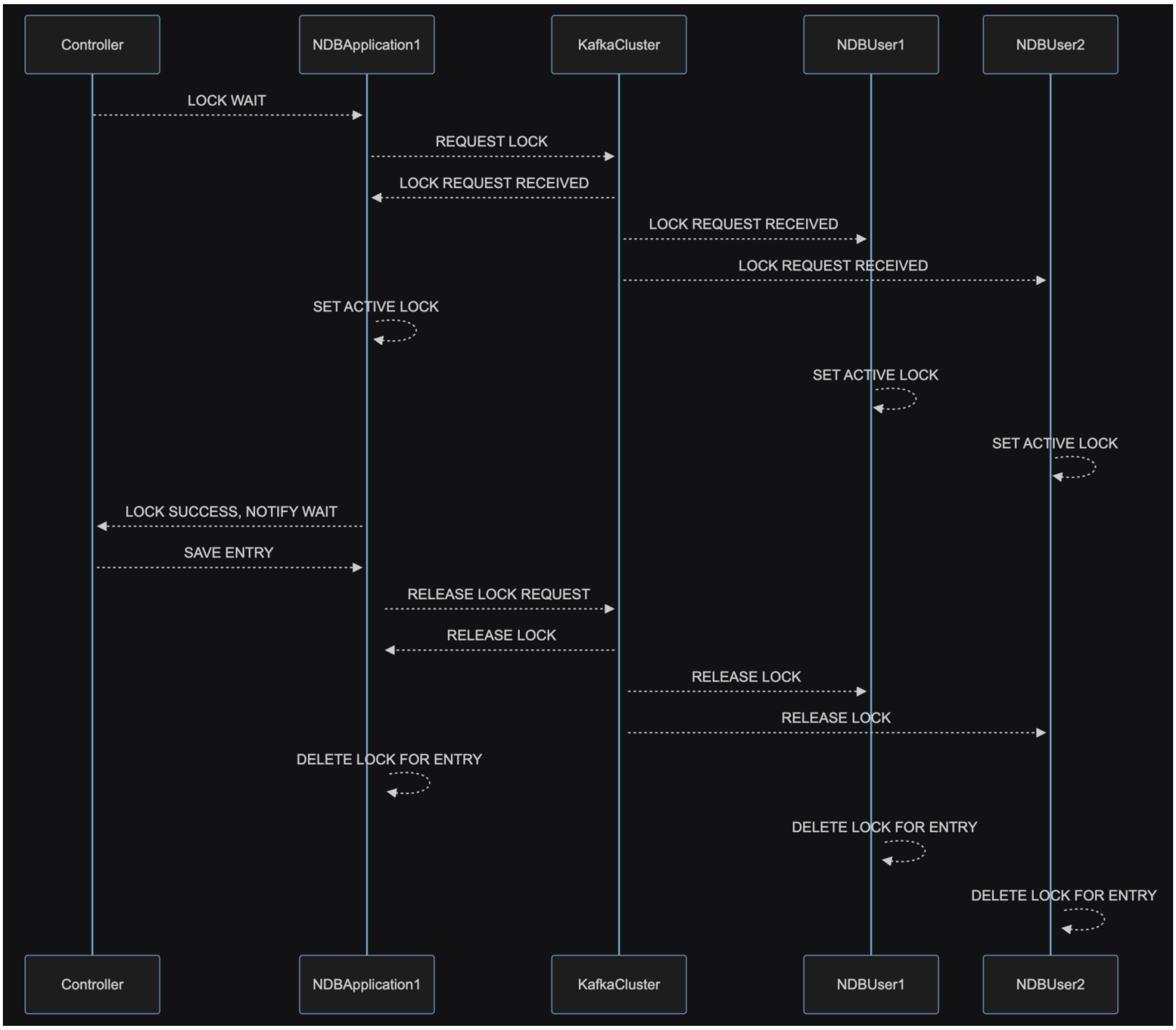
```
com.nucleodb.library.mqs.local.LocalConfiguration
```

Locks

Cluster Locks

Cluster wide DataEntry locks.

You can lock a DataEntry by specifying in the copy step that you want to lock the entry. This will send a request to lock the entry to all of the running instances of NucleoDB until that entry has been saved (Or a timeout of 1 second).



```
BookDE book = (BookDE)nucleoDB.getTable(BookDE.class).get("name", "test").iterator().next();

// true indicates a cluster wide lock
// the thread will wait until the lock is successful for this specific request
BookDE bookWritable = book.copy(BookDE.class, true);

// entry is now locked for the entire cluster
// any changes will be thread safe
bookWritable.getData().setName("Odyssey");

// save to cluster.
// NDB will release the lock and other locks will have a chance to successfully lock
nucleoDB.getTable(BookDE.class).saveSync(bookWritable);
```

```
sequenceDiagram
  Controller-->> NDBApplication1: LOCK WAIT
  NDBApplication1-->>KafkaCluster: REQUEST LOCK
  KafkaCluster-->>+NDBApplication1: LOCK REQUEST RECEIVED
  KafkaCluster-->>+ NDBUser1: LOCK REQUEST RECEIVED
  KafkaCluster-->>+ NDBUser2: LOCK REQUEST RECEIVED
  NDBApplication1 -->>+ NDBApplication1: SET ACTIVE LOCK
  NDBUser1 -->>+ NDBUser1: SET ACTIVE LOCK
  NDBUser2 -->>+ NDBUser2: SET ACTIVE LOCK
  NDBApplication1-->>+Controller: LOCK SUCCESS, NOTIFY WAIT
  Controller-->>+NDBApplication1: SAVE ENTRY
  NDBApplication1-->>+KafkaCluster: RELEASE LOCK REQUEST
  KafkaCluster -->>+NDBApplication1: RELEASE LOCK
  KafkaCluster -->>+NDBUser1: RELEASE LOCK
  KafkaCluster -->>+NDBUser2: RELEASE LOCK
```

```
NDBApplication1 -->>+NDBApplication1:DELETE LOCK FOR ENTRY
NDBUser1 -->>+NDBUser1: DELETE LOCK FOR ENTRY
NDBUser2 -->>+NDBUser2: DELETE LOCK FOR ENTRY
```

Spring Data Repository Library

Getting Started

NucleoDB Spring Repository Library makes it easier to use NucleoDB.

Installation

Dependencies

- Kafka Cluster
  - /docker/kafka/docker-compose.yml

Import library

```
repositories {
    mavenCentral()
    maven { url "https://nexus.synload.com/repository/maven-repo-releases/" }
}
dependencies {
    implementation 'com.nucleodb:spring:3.3.49'
}
```

Initializing DB

```

@SpringBootApplication
@EnableNDBRepositories(
        dbType = NucleoDB.DBType.NO_LOCAL, // does not create a disk of current up-to-date version of DB
        // Feature: Read To Time, will read only changes equal to or before the date set.
        //readToTime = "2023-12-17T00:42:32.906539Z",
        scanPackages = {
                "com.package.string" // scan for @Table classes
        },
        basePackages = "com.package.string.repos"
)
class Application{
  public static void main(String[] args) {
    SpringApplication.run(Application.class);
  }
}
```

DataEntry model files

```


import java.io.Serializable;


@Table(tableName = "author", dataEntryClass = AuthorDE.class)
public class Author implements Serializable {
  @Index
  String name;
  public Author(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }


  public void setName(String name) {
    this.name = name;
  }
}


public class AuthorDE extends DataEntry<Author>{


}
```

Repository for DataEntry

```
@Repository
public interface AuthorDataRepository extends NDBDataRepository<AuthorDE, String>{
  Set<AuthorDE> findByNameAndKey(String name, String key);
  Author findByName(String name);
  void deleteByName(String name);
}
```

Connection model files

```
@Conn("CONNECTION_BETWEEN_DE")
public class ConnectionBetweenDataEntryClasses extends Connection<ConnectingToDataEntryDE,
ConnectingFromDataEntryDE>{
  public ConnectionBetweenDataEntryClasses() {
  }

  public ConnectionBetweenDataEntryClasses(ConnectingFromDataEntryDE from, ConnectingToDataEntryDE to)
{
    super(from, to);
  }

  public ConnectionBetweenDataEntryClasses(ConnectingFromDataEntryDE from, ConnectingToDataEntryDE to,
Map<String, String> metadata) {
    super(from, to, metadata);
  }
}
```

Repository for Connections

```
@Repository
public interface ConnectionRepository extends NDBConnRepository<ConnectionBetweenDataEntryClasses,
String, ConnectingFromDataEntryDE, ConnectingFromDataEntryDE>{


}
```

Connection Repository

Connection repository integrates with the spring data repository. This enables your application to lookup based on either the destiny or the originating DataEntry.

Definition

```
@Repository
public interface AuthoredConnectionRepository
  extends NDBConnRepository<AuthoredConnection, String, BookDE, AuthorDE>{


}
```

This Repository definition contains not only the Connection (AuthoredConnection) but also the originating DataEntry (AuthorDE) and the destination DataEntry (BookDE).

Usage

```
@RestController
public class AnimeController{


  @Autowired
  AuthorDataRepository authorRepository;


  @Autowired
  AuthoredConnectionRepository authoredConnectionRepository;


  @GetMapping("/authored/{name}")
  public Set<AuthoredConnection> getByName(@PathVariable String name){
    Optional<AuthorDE> byName = authorRepository.findByName(id);
    if(byName.isPresent()){
      return authoredConnectionRepository.getByFrom(byName.get());
    }
    return null;
  }
  @DeleteMapping("/authored/{name}")
  public void deleteByName(@PathVariable String name){
    Optional<AuthorDE> byName = authorRepository.findByName(id);
    if(byName.isPresent()){
      authoredConnectionRepository.getByFrom(byName.get()).foreach(authored-
>authoredConnectionRepository.delete(authored));
    }
    return null;
  }
  @UpdateMapping("/authored/{name}/{country}")
  public void updateCountryByName(@PathVariable String name, @PathVariable String country){
    Optional<AuthorDE> byName = authorRepository.findByName(id);
    if(byName.isPresent()){
      authoredConnectionRepository.getByFrom(byName.get()).foreach(authored->{
        authored.setCountry(country);
        authoredConnectionRepository.save(authored);
      });
    }
    return null;
  }
}
```

Data Repository

Simplifies the retrieval and saving of data entries into the NucleoDB database.

Definition

```
@Repository
public interface AuthorDataRepository extends NDBDataRepository<AuthorDE, String>{
  Optional<AuthorDE> findByNameAndKey(String name, String key);
  Optional<AuthorDE> findByName(String name);
  void deleteByName(String name);
}
```

Usage

```
@RestController
public class AnimeController{

  @Autowired
  AuthorDataRepository authorRepository;



  @GetMapping("/get/{name}")
  public Optional<AuthorDE> getByName(@PathVariable String name){
    return authorRepository.findByName(id);
  }
  @DeleteMapping("/get/{name}")
  public void deleteByName(@PathVariable String name){
    Optional<AuthorDE> byName = authorRepository.findByName(id);
    if(byName.isPresent()){
      authorRepository.delete(byName.get());
    }
    return null;
```

```
   }
   @UpdateMapping("/get/{name}/{genre}")
   public void updateCountryByName(@PathVariable String name, @PathVariable String genre){
     Optional<AuthorDE> byName = authorRepository.findByName(id);
     if(byName.isPresent()){
       AuthorDE author = byName.get();
       author.getData().setGenre(genre);
       authorRepository.save(author);
     }
   }
 }
```

EventListener

Data Entry

Code Sample

```
@RestController
public class BookController{
  @EventListener
  public void bookCreatedEvent(DataEntryCreatedEvent<BookDE> bookCreated){
    Serializer.log("Book created "+bookCreated.getDataEntry().getData().getName());
  }
}
```

Events

- DataEntryCreatedEvent
- DataEntryUpdatedEvent
- DataEntryDeletedEvent

Connection

Code Sample

```
@RestController
public class BookController{
  @EventListener
  public void bookConnectionCreatedEvent(ConnectionCreatedEvent<AuthoredConnection> authored){
    Serializer.log("Authored connection created "+authored.getConnection().getMetadata().get("rating"));
  }
}
```

Events

- ConnectionCreatedEvent
- ConnectionUpdatedEvent
- ConnectionDeletedEvent

Source Code

The source code can be found on Github here https://github.com/NucleoTeam/NucleoDB