UNIVERSITY OF LEICESTER

Department of Informatics University of Leicester

CO7217 – Agile Cloud Automation

Coursework Re-sit

Name: JOHNSON AZUBUIKE ANIGBO

Student Number: 209052408

Course start Date: 19th February 2024

Note:

1. The readme. md has details on how to run code and the dependencies needed.

Exercise 1

I used a data set for currency conversion to the dollar for all the currencies in the world. So, I performed the following queries.

A. **Data Selection:** This currency conversion displays all the currencies' rates to the dollar. So here is the code for this query. Each country's currencies are represented with an abbreviation as a code. After all the data is retrieved from the database it is then formatted and represented in a table.

```
static String allDocumentsQuery(MongoCollection<Document> collection) {
def formattedTable = new StringBuilder()
    formattedTable.append(String.format("| %-20s | %-20s |\n", "Code", "Rate"))
    formattedTable.append("|----------------------|----------------------|\n")

    def allDocuments = collection.find()
    allDocuments.each { doc ->
           def code = doc.get("code")
           def rate = doc.get("rate")
           formattedTable.append(String.format("| %-20s | %-20s |\n", code, rate))

    }
return formattedTable.toString()
}
```

Here is the code for testing this query.

```
@Test
void testAllDocumentsQuery() {
    // Insert test data into the collection for this test
    collection.deleteMany(new Document())
    Document document1 = new Document("code", "SampleCode1").append("rate", 10.5)
    Document document2 = new Document("code", "SampleCode2").append("rate", 8.2)
    collection.insertMany([document1, document2])
    // Perform the test
    def result = ExerciseOne.allDocumentsQuery(collection)

    println(result)

    // Assert the results
    assertTrue(result.contains("| Code                 | Rate                 |"))
    assertTrue(result.contains("|----------------------|----------------------|"))
    assertTrue(result.contains("| SampleCode1          | 10.5                 |"))
    assertTrue(result.contains("| SampleCode2          | 8.2                  |"))
}
```

The result for this query is in Appendix A of this report.

B. **Data Projection**: This gives the data projection for this query.

```
static String projectedDocumentsQuery(MongoCollection<Document> collection) {
    def formattedTable = new StringBuilder()
    formattedTable.append(String.format("| %-20s | %-20s |\n", "Code", "Rate"))
    formattedTable.append("|----------------------|----------------------|\n")


    def projectionDocument = new Document("code", 1).append("rate", 1).append("_id", 0)
    def projectedDocuments = collection.find().projection(projectionDocument)
    projectedDocuments.each { doc ->
        def code = doc.get("code")
        def rate = doc.get("rate")
        formattedTable.append(String.format("| %-20s | %-20s |\n", code, rate))
    }
    return formattedTable.toString()

}
```

Here is the test for this query.

```
@Test
void testProjectedDocumentsQuery() {
    // Insert test data into the collection for this test
    collection.deleteMany(new Document())

    Document document1 = new Document("code", "SampleCode1").append("rate", 10.5)
    Document document2 = new Document("code", "SampleCode2").append("rate", 8.2)
    Document document3 = new Document("code", "SampleCode3").append("rate", 7.0)

    collection.insertMany([document1, document2, document3])

    // Perform the test
    def result = ExerciseOne.projectedDocumentsQuery(collection)

    // Print the result for manual inspection
    println(result)

    // Assert the results
    assertTrue(result.contains("| Code                 | Rate                 |"))
    assertTrue(result.contains("|----------------------|----------------------|"))
    assertTrue(result.contains("| SampleCode1          | 10.5                 |"))
    assertTrue(result.contains("| SampleCode2          | 8.2                  |"))
    assertTrue(result.contains("| SampleCode3          | 7.0                  |"))
}
```

C. **Data Filtering:** This query displays countries whose conversion rate to the dollar is above five (5).

```
static String filteredDocumentsQuery(MongoCollection<Document> collection) {
    def formattedTable = new StringBuilder()
    formattedTable.append(String.format("| %-20s | %-20s |\n", "Code", "Rate"))
    formattedTable.append("|----------------------|----------------------|\n")


    def filterDocument = new Document("rate", new Document("\$gt", 5.0))
    def filteredDocuments = collection.find(filterDocument)
    filteredDocuments.each { doc ->
        def code = doc.get("code")
        def rate = doc.get("rate")
        formattedTable.append(String.format("| %-20s | %-20s |\n", code, rate))
    }
    return formattedTable.toString()

}
```

Here is the code that performs the test for this query.

```
@Test
void testFilteredDocumentsQuery() {

    collection.deleteMany(new Document())
    Document document1 = new Document("code", "SampleCode1").append("rate", 10.5)
    Document document2 = new Document("code", "SampleCode2").append("rate", 8.2)
    Document document3 = new Document("code", "SampleCode2").append("rate", 4.0)
    collection.insertMany([document1, document2,document3])



    // Perform the test
    def result = ExerciseOne.filteredDocumentsQuery(collection)

    // Assert the results
    assertTrue(result.contains("| Code                  | Rate                  |"))
    assertTrue(result.contains("|-----------------------|-----------------------|"))
    assertTrue(result.contains("| SampleCode1           | 10.5                  |"))
    assertTrue(result.contains("| SampleCode2           | 8.2                   |"))

}
```

D. **Data Combination:** This query gets an average for the country's code first character, groups countries that have the same alphabet as one group, and calculates their average.

```
static String groupedDocumentsQuery(MongoCollection<Document> collection) {
    def formattedTable = new StringBuilder()
    formattedTable.append(String.format("| %-20s | %-20s |\n", "Id", "Average Rate"))
    formattedTable.append("|----------------------|----------------------|\n")


    def result = ""
    def groupDocument = new Document("_id", new Document("\$substr", ["\$code", 0, 1])).append("averageRate", new Document("\$avg", "\$rate"))
    def groupedDocuments = collection.aggregate([new Document("\$group", groupDocument)])
    groupedDocuments.each { doc ->
        formattedTable.append(String.format("| %-20s | %-20s |\n", doc._id, doc.averageRate))

        }
        return formattedTable.toString()

}
```

Here is the testing code for this query.

```
@Test
void testGroupedDocumentsQuery() {

collection.deleteMany(new Document())
Document document1 = new Document("code", "SampleCode1").append("rate", 10.5)
Document document2 = new Document("code", "SampleCode2").append("rate", 8.2)
Document document3 = new Document("code", "SampleCode3").append("rate", 7.0)
Document document4 = new Document("code", "SampleCode4").append("rate", 5.5)

collection.insertMany([document1, document2,document3, document4 ])

    // Perform the test
    def result = ExerciseOne.groupedDocumentsQuery(collection)

    // Print the result for manual inspection
    println(result)

    // Assert the results
    assertTrue(result.contains("| Id                    | Average Rate          |"))
    assertTrue(result.contains("|-----------------------|-----------------------|"))
    assertTrue(result.contains("| S                     | 7.8                   |"))
}
```

Exercise 2

The currency exchange rate system that I used in exercise one consists of over a hundred counties. The exchange rate of many countries seems to be changing at a fast pace. So, for users from various countries trying to access the exchange rate of their countries, it will require replication and sharding to handle such a scale. Therefore, let's delve into the scalability implications of using MongoDB in the context of a currency exchange rate system, with a specific focus on replication and sharding.

Advantage Of Replication

1. High Availability for Real-Time Rate: For currency exchange rate systems, real-time update of the system is important. Because the exchange rate of any country is going to be changing at any moment, therefore, the system must be up and running every moment. So, MongoDB's replication ensures constant availability of the system to take updates on these changes. If the primary server is overwhelmed by write or update requests one of the secondary servers should be able to handle such requests.
2. Read Scalability for Concurrent Access: Replication allows for distributing read requests over many nodes. This is specifically useful in a currency exchange system where many users might be querying rates simultaneously. Read scalability ensures efficient and quick access to exchange rate data during peak usage.

Disadvantages Of Replication

1. Consistency Trade-off: MongoDB's default eventual consistency might be acceptable in a currency exchange rate system where the latest rates are crucial. However, ensuring that eventual consistency aligns with the application's requirements is essential, as strict consistency might be necessary for certain operations.

2. Configuration Complexity: Setting up and configuring MongoDB replication involves considerations such as choosing the appropriate replica set members and configurations. While the MongoDB documentation provides guidance, a thorough understanding is crucial to optimizing the replication setup for the specific needs of the currency exchange rate system.

Conclusion

The advantages and disadvantages should be evaluated based on the specific requirements of the application, emphasizing the need for real-time access, high availability, and efficient handling of growing data volumes. Careful configuration and optimization are essential for ensuring the scalability of the system in the dynamic context of currency exchange rates.

Exercise 3

Aim of the DSL

1. Problem Domain:
   The problem domain involves managing user information within a cloud-based system. Key tasks include establishing connections to a cloud database, defining the schema for database collections, and implementing CRUD (Create, Read, Update, Delete) operations using MongoDB. The objective is to generate code that specifically interacts with MongoDB to facilitate seamless user data management in a cloud environment. This entails handling database connections, structuring data using specified schemas, and executing CRUD operations in compliance with MongoDB conventions. The solution aims to streamline user information management within a cloud-based infrastructure, utilizing MongoDB as the underlying database technology.

2. Scope of the Project:
   The project's scope includes developing a DSL to automate tasks related to user data management, such as database connection, schema definition, and CRUD operations. The DSL will generate code to interact with a cloud database and streamline user information handling.
3. Key Concepts:
   a. Connection String:
      - Intrinsic Properties: database name
   b. User Data Schema:
      - Intrinsic Properties: Data types, constraints, relationships, references etc
   c. CRUD Operations:
      - Intrinsic Properties: Operations (Create, Read, Update, Delete), query inputs.

Objectives of the DSL:

a. Connection String Generation:
   Automate the generation of connection strings based on user-provided parameters, ensuring secure and efficient connections to the cloud database.
b. Schema Definition:
   Allow users to define the schema for user data using a concise DSL syntax. This includes specifying data types, constraints, and relationships.
c. CRUD Operation Scripts:
   Generate scripts for CRUD operations (Create, Read, Update, Delete) based on the defined schema. This automates the process of interacting with the database for user data management.

Examples and Potential Benefits:

1. Example 1: Connection String Generation

- Manual Approach: Developers manually construct connection strings, which can lead to errors and security vulnerabilities.
- DSL Automation: Using the DSL, developers can define connection parameters concisely, and the DSL generates secure connection strings. This reduces the risk of misconfigurations and improves security.

2. Example 2: Schema Definition

- Manual Approach: Creating and modifying database schemas is a complex process involving SQL scripts or manual changes in a database management tool.
- DSL Automation: The DSL allows developers to define the user data schema in a human-readable syntax. This schema definition is then translated into database-specific scripts, streamlining the schema management process.

3. Example 3: CRUD Operation Scripts

- Manual Approach: Writing SQL queries or using an ORM tool to perform CRUD operations requires detailed knowledge of database interactions.
- DSL Automation: With the DSL, developers express CRUD operations in a higher-level syntax. The DSL generates optimized database queries or ORM calls, reducing development time and potential errors.

The DSL for cloud-based user information management streamlines and automates tasks related to connecting to a database, defining data schema, and performing CRUD operations. This results in increased developer productivity, reduced chances of errors, and a more efficient process for managing user data in a cloud environment. The DSL provides a concise and domain-specific way to interact with cloud databases for user information storage and retrieval.

Exercise 4

A. An xtext grammar

This DSL is focused on being used in node JS applications to call Mongo DB APIs. Here is a brief explanation of the rules defined in the text DSL

I. Model: Defines the main Model element, which consists of a ConnectionStatement and a list of OperationStatements.

II. OperationStatements: Describes the syntax for a connection statement with properties like Host, Port, Database, Username, Password, and Options.

III. MakeStatement: Defines the syntax for creating a collection for MongoDB collection with optional dictionary entries.

IV. DictionaryEntry: Specifies the syntax for dictionary entries, which can have a name, a field type (String, Integer, Boolean), an optional "required" status, and an optional "unique" key. It also allows references to other collections.

So MakeStatement and DictionaryEntry is for creating schema for the database

V. OtherOperationStatement: Describes the syntax for task operations, including types like GET, DELETE, INSERT, and UPDATE. It defines the collection, input for these operations, and optional update for the UPDATE operation.

VI. TaskDictionaryEntry: Specifies the syntax for collection entries or attributes used in task operations, which can have a value name and either a string or an integer value.

```
grammar org.xtext.example.mydsl1.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl1/MyDsl"


MongoDB:
      models += Model*;


Model:
  connection=ConnectionStatement
  statements+=OperationStatement*;

OperationStatement:
  schema=MakeStatement operations+=OtherOperationStatement*;

ConnectionStatement:
  'DatabaseParameters' '{'
  'Connection' connectionString=STRING ','&
      'database' database=STRING
  '}';



MakeStatement:
  'Collection' name=ID
  '{'
    (entries+=DictionaryEntry (',' entries+=DictionaryEntry)*)?
  '}';

DictionaryEntry:
  name=ID ':' fieldType=FieldType (status?="required")? (uniqueKey?='unique')?
  | name=ID ':' 'ref' refDictionary=[MakeStatement | ID];

OtherOperationStatement:
  'task:'( '{'
    'operation:' type=Operation ','
    'collection:' collection=[MakeStatement]
    'input:' '{'
      (entries+=TaskDictionaryEntry (',' entries+=TaskDictionaryEntry)*)?
    '}'
  '}') | 'task:'( '{'
    'operation:' type='UPDATE,'
    'collection:' collection=[MakeStatement]
    'input:' '{'
      (entries+=TaskDictionaryEntry (',' entries+=TaskDictionaryEntry)*)?
    '}'
    (',' "update:"'{'
      (entries+=TaskDictionaryEntry (',' entries+=TaskDictionaryEntry)*)?
    '}')?
  '}');

TaskDictionaryEntry:
  valueName=ID ':' (valueString=STRING | valueInt=INT);

Operation:
  'GET' | 'DELETE' | 'INSERT';

FieldType:
  'String' | 'Number' | 'Boolean'|'Date';
```

There are also validation rules for this DSL and they are as follows:

i. The collection name must start with an uppercase letter

```
@Check
public void validateCollectionName(MakeStatement entry) {
    // Check the dictionary name

    String name = entry.getName();
    if (name != null && !name.isEmpty()) {
        char firstChar = name.charAt(0);
        if (!Character.isUpperCase(firstChar)) {
            warning("collection name should start with an uppercase letter",
                    MyDslPackage.Literals.MAKE_STATEMENT__NAME);
        }
    }
}
```

ii. The collection attribute name must start with a lowercase letter.

```
@Check
public void checkAttributeNameStartsWithLowercase(DictionaryEntry attr) {
    if (!Character.isLowerCase(attr.getName().charAt(0))) {
        warning("collection attribute name should start with a lowercase",
                MyDslPackage.Literals.DICTIONARY_ENTRY__NAME);
    }
}
```

iii. The collection must not reference itself.

```
@Check
public void checkNoSelfReference(MakeStatement make) {
    for (DictionaryEntry entry : make.getEntries()) {
        if (entry.getRefDictionary() != null && entry.getRefDictionary().getName().equals(make.getName())) {
            error("A collection cannot reference itself.", MyDslPackage.Literals.MAKE_STATEMENT__NAME);
        }
    }
}
```

B. Xtext example

I. Example 1

```
  DatabaseParameters{
  Connection
"mongodb+srv://anigbojohnsona:Sonship123@cluster0.kybkuyh.mongodb.net/FTMP?retr
yWrites=true&w=majority" ,
   database "FTMP"
  }

Collection User {
  id: Number  unique,
  username: String required unique

}
Collection Order {
  id: Number  required unique ,
  price: String required unique,
  userOrder: ref User
}

task: {
  operation: INSERT,
  collection: User
  input: {
    id : 1,
    username:"johnson"
  }
}

task: {
  operation: INSERT,
  collection: User
  input: {
    id : 2,
    username:"thompson"
  }
}
task: {
  operation: INSERT,
  collection: Order
  input: {
    id : 1,
    price:12,
    userOrder:1
  }
}

task: {
  operation: INSERT,
  collection: Order
  input: {
    id : 2,
    price:30,
     userOrder:2
  }
}

task: {
  operation: GET,
  collection: Order
  input: {
    id : 1
  }
}
```

II. Example 2

```
  DatabaseParameters{
  Connection
"mongodb+srv://anigbojohnsona:Sonship123@cluster0.kybkuyh.mongodb.net/FTMP?retr
yWrites=true&w=majority" ,
   database "FTMP"
  }


Collection MyCollection {
    entry1: String required unique,
    entry2: Number,
    entry3: Boolean
}

task: {
    operation: GET,
    collection: MyCollection
    input: {
        entry1: "Value1",
        entry2: 42
    }
}

task: {
    operation: DELETE,
    collection: MyCollection
    input: {
        TaskEntry1: "Value1"
    }
}

task: {
    operation: INSERT,
    collection: MyCollection
    input: {
         entry1: "Value2",
         entry2: 22
    }
}

task: {
    operation: UPDATE,
    collection: MyCollection
    input: {
        entry1: "Value2"
    },
    update: {
        entry2: 456
    }
}
```

Exercise 5

i. The code initializes an Xtext injector and resource set, loads a DSL file, extracts its root MongoDB object, traverses its models, generates MongoDB-related code, writes it to an output file, and prints a confirmation message. The DSL file path, output file, and traversal function are specified. The generated code includes MongoDB schema creation and connection setup based on DSL-defined structures.

```
static void main(String[] args) {

    def injector = new MyDslStandaloneSetup().createInjectorAndDoEMFRegistration()
    // obtain a resource set from the injector

    // obtain a resourceset from the injector
    XtextResourceSet resourceSet = injector.getInstance(XtextResourceSet.class)

    def path = "C:/Users/THOMPSON ANIGBO/eclipse-workspace/org.xtext.example.mydsl1/resources/example1.mydsl1" // Adj

    // load a resource by URI, in this case from the file system
    Resource resource = resourceSet.getResource(URI.createFileURI(path), true)

    def mongo =(MongoDB)resource.getContents().get(0)

    /**
     * TODO: Use a traversal strategy and call code templates
     */

    def text = traverse(mongo.models)

    // Write the generated code to a file
    // Adjust the output file path as needed
    def output = new File("C:/Users/THOMPSON ANIGBO/eclipse-workspace/org.xtext.example.mydsl1/resources/output.js")
    output.delete()
    output.createNewFile()
    output << text
    println("Generated code written to: ${output.absolutePath}")
}
```

ii. The code defines a static method, 'traverse,' that iterates through a list of EObjects, and invoking a 'generatePatterns' method. The method builds a string by concatenating the results of the 'generatePatterns' method for each object and returns the accumulated text.

```
def static String traverse(List<EObject> list) {
    String text = ''
    for (obj in list) {
        println(obj)
        println("Traversing element: ${obj.eClass().name}")
        text += generatePatterns(obj)
    }
    return text
}
```

iii. The code defines a static method, 'generatePatterns,' that takes an EObject as input, initializes an empty string, and employs a switch statement to handle different EObject types. Depending on the type, it invokes specific methods to generate MongoDB-related code. For instance, it generates connection code for ConnectionStatement or schema code for OperationStatement. The method accumulates the generated code in a string and returns it.

```
def static String generatePatterns(EObject element) {

    def text = ""
    switch (element) {
        case Model:
        text += generateMongoDBConnectionCode(element. connection)
        text += traverse(element.statements)
        break;
        case OtherOperationStatement:
        text += traverse(element. statements)
        break;
        case ConnectionStatement:
        text += generateMongoDBConnectionCode(element)
        break;

        case OperationStatement:
        text += generateMongoDBSchema(element.schema)
        break;
    }
    return text
}
```

iv. This code defines a static method, generateMongoDBSchema, for generating MongoDB schema code based on the provided MakeStatement DSL element. It creates a string representation of a MongoDB schema, including the collection name derived from the MakeStatement's name. The code iterates over the dictionary entries within the MakeStatement to generate corresponding schema fields using the generateSchemaField method. Finally, it appends additional configuration options for timestamps and returns the generated code as a string.

```
def static String generateMongoDBSchema(MakeStatement schema) {

String code = """
        const ${schema.name.toLowerCase()}Schema = new mongoose.Schema({
    """

    // Generate schema fields based on dictionary entries
    for (entry in schema.entries) {
        code += generateSchemaField(entry)
    }

    code += """
        }, {
            timestamps: true,
        });
    """

    return code
}
```

v. This code defines a static method, generateMongoDBConnectionCode, responsible for generating MongoDB connection code based on the provided ConnectionStatement DSL element. It extracts the connection string and database name properties from the ConnectionStatement, then uses this information to generate JavaScript code for establishing a connection to MongoDB using the mongodb Node.js driver.

```
def static String generateMongoDBConnectionCode(ConnectionStatement connection) {

    // Access properties of ConnectionStatement
    String connectionString = connection.connectionString
    String databaseName = connection.database

    // Generate code based on extracted information
    return """

        const { MongoClient } = require('mongodb');
        const client = new MongoClient("${connectionString}",{ useNewUrlParser: true, useUnifiedTopology: true });
        let db = undefined;

        async function connect() {
            try {
                await client.connect();
                db = await client.db("${databaseName}");
                console.log("Connected successfully to server");
            } catch (error) {
                console.error(error);
            }
        }
       connect()
    """

}
```

References

Mongo DB. (2024, February 19). *Replication*. Retrieved from MongoDB Manual: https://www.mongodb.com/docs/manual/replication/