LLM served by Perplexity Labs
Hello! How can I help you?

Copy

i want to create a java project. spring boot can be used. I am an amateur at this so need your inputs.
the purpose of the project is to migrate the database from on-prem to OCI. both databases are oracle 19c.
The on-prem database at the moment has the following DBOs - tables, sequences, table partitions, indexes and constraints on the table. I guess the primary purpose of the java application could be just data load, which means DDL operations can be handled manually.

Could you provide me a high level roap map first.

LLM served by Perplexity Labs
To create a Spring Boot-based Java project that migrates data from an on-premises Oracle 19c database to Oracle Cloud Infrastructure (OCI), here is a high-level roadmap:

High-Level Roadmap

1. Understand the Requirements

- **Source and Target Databases**: Both are Oracle 19c.
- **Objects to Migrate**: Tables, sequences, partitions, indexes, and constraints.
- **Scope**: Focus on data loading; DDL operations (like schema creation) are handled manually.
- **Objective**: Build a Java-based application, using Spring Boot, to facilitate data migration.

2. Environment Setup

- **Prerequisites**:

- Install Java 17 (minimum requirement for Spring Boot 3).
- Set up IntelliJ IDEA or another IDE with Spring Boot and Flyway plugins.
- Ensure OCI access is configured, including licensing and networking (like VPN, FastConnect, or public IP for the target environment) [2] [6].

- **Database Connectivity**:
  - Configure JDBC drivers for Oracle 19c in your project.

3. Project Setup

- Use **Spring Initializr** (via the web interface or IDE) to generate a Spring Boot project:
  - Include dependencies: `Spring Data JPA`, `Flyway`, `Spring Web`, `Spring Boot DevTools`, and `Oracle JDBC Driver` [3] [10].
- Configure the `application.yml` or `application.properties` file for two databases:

```
text
```

```
spring:
  datasource:
    url: jdbc:oracle:thin:@<on-prem-database>:1521:<SID>
    username: <source_db_user>
    password: <password>
    driver-class-name: oracle.jdbc.OracleDriver

  oci-datasource:
    url: jdbc:oracle:thin:@<oci-database>:1521/<service_name>
    username: <target_db_user>
    password: <password>
    driver-class-name: oracle.jdbc.OracleDriver
```

4. Data Migration Logic

- **Read Data from On-Premise Database**:
  - Use Spring Data JPA or plain JDBC to query tables and retrieve partitions, indexes, and sequences [3] [7].
- **Write Data to OCI Database**:
  - Insert data into corresponding OCI tables using batch operations to optimize performance.

- Implement transactional support to maintain consistency and handle migration failures.

5. Optional: Use Flyway for Schema Versioning

- Although you intend to manage DDL manually, Flyway can help version track schema for better control:
  - Add Flyway to the project.
  - Store migration scripts in the `src/main/resources/db/migration` folder, following Flyway's naming conventions (e.g., `V1__initial_schema.sql`) [7] [10].
  - Configure `application.properties`:

```text
spring.flyway.locations=classpath:db/migration
```

6. Testing the Application

- Use mock or cloned datasets for testing.
- Simulate migrations between test and dev instances of Oracle 19c.
- Implement logging to track errors and migration progress.

7. Deploy to OCI

- Package the project into a JAR/WAR file using Maven or Gradle.
- Deploy it on OCI services, such as Oracle Cloud Compute or Kubernetes (if needed), and configure runtime properties [6].

8. Run Data Migration

- Execute the application with source and target database information.
- Monitor migration performance, and evaluate data integrity post-migration using Oracle database checks (e.g., `DBMS_STATS` and `DBA_OBJECTS` queries) [2] [5].

9. Post-Migration Validation

- Cross-check row counts, indexes, and constraints between databases.
- Ensure all sequences and partitions are recreated correctly.

10. Documentation and Maintenance

- Document the migration process.

- Maintain scripts and application logic for incremental data migrations if needed later.

By following these steps, you can create a robust and scalable Java-Spring Boot project for database migration from an on-prem Oracle 19c database to OCI.

Copy Ask Perplexity

please show me, how the project tree in java code base will look like? Also let me know, maybe in one or two lines the purpose of each class that's in the project tree. Please feel free to add other relevant details that could help me in any way.

LLM served by Perplexity Labs

Below is an example of how the project tree in a Java-based Spring Boot application for database migration might look, along with the purpose of each class and key files in the structure:

Proposed Project Structure

```
text

src
├── main
│   ├── java
│   │   └── com.example.databasemigration
│   │       ├── config
│   │       │   ├── DataSourceConfig.java
│   │       │   └── FlywayConfig.java
│   │       ├── models
│   │       │   ├── TableA.java
│   │       │   ├── TableB.java
│   │       │   └── SequenceInfo.java
│   │       ├── repository
│   │       │   ├── TableARepository.java
│   │       │   ├── TableBRepository.java
│   │       │   └── SequenceRepository.java
│   │       ├── service
│   │       │   ├── MigrationService.java
│   │       │   └── ValidationService.java
│   │       ├── controller
│   │       │   └── MigrationController.java
│   │       └── DatabaseMigrationApplication.java
```

```
│   ├── resources
│   │   ├── application.yml
│   │   └── db
│   │       └── migration
│   │           ├── V1__create_table_a.sql
│   │           ├── V2__create_table_b.sql
│   │           └── V3__create_sequences.sql
└── test
    └── java
        └── com.example.databasemigration
            └── DatabaseMigrationApplicationTests.java
```

Purpose of Each Class

Configuration

- `DataSourceConfig.java`
  - Configures separate data sources for the on-premises and OCI Oracle databases.
  - Ensures proper database connections and properties for migration.
- `FlywayConfig.java`
  - Configures Flyway for managing schema migrations, setting the location of migration scripts, and applying them to the target OCI database.

Models

- `TableA.java`, `TableB.java`
  - Represent the database tables as Java entities using JPA; these classes map the database schema to Java objects.
- `SequenceInfo.java`
  - Represents database sequences (metadata) for handling sequence synchronization during migration.

Repository

- `TableARepository.java`, `TableBRepository.java`
  - Provide CRUD operations on the respective tables using Spring Data JPA.
- `SequenceRepository.java`

- Manages database sequences for reading and writing sequence metadata.

Service

- `MigrationService.java`
  - Contains the core logic for migrating data. It reads data from the on-prem database and writes it to the OCI database in a batched and optimized manner.
- `ValidationService.java`
  - Provides end-to-end validation of migrated data, ensuring data consistency and integrity between the source and target databases.

Controller

- `MigrationController.java`
  - Exposes REST APIs for triggering data migration and checking migration status.
  - Serves as the entry point for external systems or manual invocation.

Application Class

- `DatabaseMigrationApplication.java`
  - The main class for running the Spring Boot application.

Purpose of Key Resource Files

- `application.yml`
  - Contains configuration for both data sources (on-prem and OCI), Flyway, and application settings.
- `db/migration/`
  - Stores Flyway migration scripts for schema creation. For example:
    - `V1__create_table_a.sql` : SQL script for creating `TableA` .
    - `V2__create_table_b.sql` : SQL script for creating `TableB` .
    - `V3__create_sequences.sql` : SQL script for creating sequences required in the target database.

Additional Notes

- **Migration Batches**: Use batching for data transfer to avoid memory overflows during massive data migrations.

- **Testing Setup**: The `DatabaseMigrationApplicationTests.java` file sets up integration tests for validating the migration logic against mock or test databases.
- **Error Handling**: Ensure the `MigrationService` includes robust error handling to log and retry if data migration fails for some records.
- **Security**: Protect sensitive database credentials in `application.yml` using tools like Spring Cloud Config or environment variables.

This structure ensures modularity, maintainability, and testability while catering to the specific needs of the migration project.

Copy Ask Perplexity

Could you please provide the source code also for a sample project?

LLM served by Perplexity Labs

Here is the source code for a basic Spring Boot database migration project. It includes a subset of files to give you a starting point for implementation.

Project: Database Migration Tool

Primary Configuration and Application Files

1. `DatabaseMigrationApplication.java`
   Main entry point for the Spring Boot application:

```java
package com.example.databasemigration;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DatabaseMigrationApplication {
    public static void main(String[] args) {
        SpringApplication.run(DatabaseMigrationApplication.class,
args);
    }
}
```

2. `DataSourceConfig.java`
   Configures the data sources for both on-prem and OCI databases:

```java
package com.example.databasemigration.config;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
public class DataSourceConfig {

    @Bean(name = "onPremDataSource")
    public DataSource onPremDataSource() {
        return DataSourceBuilder.create()
                .url("jdbc:oracle:thin:@<on-prem-url>:1521:<SID>")
                .username("<source_user>")
                .password("<source_password>")
                .driverClassName("oracle.jdbc.OracleDriver")
                .build();
    }

    @Bean(name = "ociDataSource")
    public DataSource ociDataSource() {
        return DataSourceBuilder.create()
                .url("jdbc:oracle:thin:@<oci-
url>:1521/<service_name>")
                .username("<target_user>")
                .password("<target_password>")
                .driverClassName("oracle.jdbc.OracleDriver")
                .build();
    }
}
```

3. `application.yml`
   Configure application properties for both data sources:

```text
spring:
  datasource:
    url: jdbc:oracle:thin:@localhost:1521:orcl
    username: user
    password: pass
    driver-class-name: oracle.jdbc.OracleDriver

  flyway:
    enabled: true
    locations: classpath:db/migration
```

Model Classes

4. `TableA.java`
   An entity class for mapping a sample table:

```java
package com.example.databasemigration.models;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class TableA {

    @Id
    private Long id;
    private String name;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
```

```
        return name;
    }

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

Repository Layer

5. `TableARepository.java`
   Repository interface for accessing `TableA`:

```java
package com.example.databasemigration.repository;

import com.example.databasemigration.models.TableA;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TableARepository extends JpaRepository<TableA, Long> {
}
```

Service Layer

6. `MigrationService.java`
   Core logic for migrating data:

```java
package com.example.databasemigration.service;

import com.example.databasemigration.models.TableA;
import com.example.databasemigration.repository.TableARepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.sql.Connection;
```

```
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

@Service
public class MigrationService {

    @Autowired
    private DataSource onPremDataSource;

    @Autowired
    private TableARepository tableARepository;

    public void migrateTableA() {
        try (Connection sourceConnection =
onPremDataSource.getConnection();
             Statement statement =
sourceConnection.createStatement()) {

            String query = "SELECT id, name FROM TableA";
            ResultSet resultSet = statement.executeQuery(query);

            List<TableA> targetData = new ArrayList<>();
            while (resultSet.next()) {
                TableA tableA = new TableA();
                tableA.setId(resultSet.getLong("id"));
                tableA.setName(resultSet.getString("name"));
                targetData.add(tableA);
            }

            // Save data to OCI using JPA Repository
            tableARepository.saveAll(targetData);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Controller Layer

7. `MigrationController.java`
   REST API to trigger the migration:

```java
package com.example.databasemigration.controller;

import com.example.databasemigration.service.MigrationService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MigrationController {

    @Autowired
    private MigrationService migrationService;

    @GetMapping("/migrate/tableA")
    public String migrateTableA() {
        migrationService.migrateTableA();
        return "Migration of TableA completed successfully!";
    }
}
```

Flyway Migration Script

8. `src/main/resources/db/migration/V1__create_table_a.sql`
   Example schema creation script:

```sql
CREATE TABLE TableA (
    id NUMBER PRIMARY KEY,
    name VARCHAR2(100)
);
```

Testing

9. `DatabaseMigrationApplicationTests.java`
   Basic integration test:

```java
package com.example.databasemigration;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class DatabaseMigrationApplicationTests {
    @Test
    void contextLoads() {
    }
}
```

How to Run the Project

1. Build and run the Spring Boot project (`DatabaseMigrationApplication`) using your IDE or command line (`mvn spring-boot:run`).
2. Ensure the source and target databases are reachable.
3. Access the API endpoint, e.g., `http://localhost:8080/migrate/tableA` , to trigger data migration.

This sample is simplistic and should be extended for production with features like error handling, batch processing, and validation.

Copy Ask Perplexity

how will I integrate batch processing, and validation?

LLM served by Perplexity Labs

To integrate **batch processing** and **validation** into your Java-based Spring Boot application for Oracle database migration, follow these steps:

1. Batch Processing Integration

Spring Batch is ideal for managing batch jobs like data migration. Here's how you can integrate it:

Add Spring Batch Dependency

Add the following dependency in your `pom.xml`:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-batch</artifactId>
</dependency>
```

Configuration for Spring Batch

Create a `BatchConfig` class to define batch jobs, steps, readers, processors, and writers:

```java
@Configuration
@EnableBatchProcessing
public class BatchConfig {

    @Bean
    public Job migrationJob(JobRepository jobRepository, Step
migrationStep) {
        return new JobBuilder("migrationJob", jobRepository)
                .start(migrationStep)
                .build();
    }

    @Bean
    public Step migrationStep(JobRepository jobRepository,
PlatformTransactionManager transactionManager,
                              JdbcCursorItemReader<TableA> reader,
JdbcBatchItemWriter<TableA> writer) {
        return new StepBuilder("migrationStep", jobRepository)
                .<TableA, TableA>chunk(100, transactionManager)
                .reader(reader)
                .processor(processor())
                .writer(writer)
                .build();
    }

    @Bean
```

```
    public JdbcCursorItemReader<TableA>
reader(@Qualifier("onPremDataSource") DataSource dataSource) {
        return new JdbcCursorItemReaderBuilder<TableA>()
                .name("tableAReader")
                .dataSource(dataSource)
                .sql("SELECT id, name FROM TableA")
                .rowMapper(new BeanPropertyRowMapper<>(TableA.class))
                .build();
    }

    @Bean
    public JdbcBatchItemWriter<TableA> writer(@Qualifier("ociDataSource")
DataSource dataSource) {
        return new JdbcBatchItemWriterBuilder<TableA>()
                .itemSqlParameterSourceProvider(new
BeanPropertyItemSqlParameterSourceProvider<>())
                .sql("INSERT INTO TableA (id, name) VALUES (:id, :name)")
                .dataSource(dataSource)
                .build();
    }

    @Bean
    public ItemProcessor<TableA, TableA> processor() {
        return item -> {
            // Include any transformation logic if needed
            item.setName(item.getName().toUpperCase());
            return item;
        };
    }
}
```

Explanation

- **Reader**: Reads data in chunks from the source database.
- **Processor**: Transforms data (e.g., converts to uppercase or validates).
- **Writer**: Writes the transformed data into the target database.
- **Job/Step**: A collection of steps that define the sequence of execution.

Triggering the Batch Job

Use a REST controller to start the batch job:

```java
@RestController
public class BatchController {

    private final JobLauncher jobLauncher;
    private final Job migrationJob;

    public BatchController(JobLauncher jobLauncher, Job migrationJob) {
        this.jobLauncher = jobLauncher;
        this.migrationJob = migrationJob;
    }

    @GetMapping("/start-batch-job")
    public String startBatchJob() {
        try {
            JobExecution execution = jobLauncher.run(migrationJob,
new JobParameters());
            return "Batch job completed with status: "
+ execution.getStatus();
        } catch (Exception e) {
            return "Batch job failed: " + e.getMessage();
        }
    }
}
```

2. Validation Implementation

Validation can be done at multiple levels, including row-level during migration and post-migration database validation.

Validation During Migration

Use an `ItemProcessor` to implement data validation:

```java
@Bean
public ItemProcessor<TableA, TableA> validationProcessor() {
    return item -> {
        if (item.getId() == null || item.getName() == null ||
item.getName().isEmpty()) {
            throw new ValidationException("Invalid data: "
```

```
+ item.toString());
            }
            return item;
        };
    }
```

Post-Migration Validation

After migration, compare source and target databases for consistency.

Row Count Validation

Use SQL queries to ensure data counts match:

```sql
SELECT COUNT(*) FROM TableA;
```

Checksum Validation

Generate checksums for rows in the source and target databases:

```sql
SELECT DBMS_CRYPTO.HASH(UTL_RAW.CAST_TO_RAW(CONCAT(id, name)), 2) AS
checksum FROM TableA;
```

Object Validation

Verify that schema objects (tables, indexes, constraints, etc.) match between source and target. Use the following queries as an example:

- **Constraints**: `SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS;`
- **Indexes**: `SELECT INDEX_NAME FROM USER_INDEXES;`

Automated Validation API

Create a service to perform validation checks:

```java
@Service
public class ValidationService {
```

sonar-pro

Ask anything...

```
    public void validateRowCounts() throws SQLException {
        String sql = "SELECT COUNT(*) FROM TableA";

        try (Connection sourceConn = sourceDataSource.getConnection();
             Connection targetConn = targetDataSource.getConnection();
             PreparedStatement sourceStmt =
sourceConn.prepareStatement(sql);
             PreparedStatement targetStmt =
targetConn.prepareStatement(sql);
             ResultSet sourceRs = sourceStmt.executeQuery();
             ResultSet targetRs = targetStmt.executeQuery()) {

            sourceRs.next();
            targetRs.next();
            if (sourceRs.getInt(1) != targetRs.getInt(1)) {
                throw new RuntimeException("Row count
mismatch detected!");
            }
        }
    }
}
```

Invoke this validation service at the end of the batch job.

Tools and Best Practices

1. **Spring Retry**: Add retry logic for transient errors during batch processing.
2. **Logging**: Log processed records and errors for debugging.
3. **Error Handling**: Use `SkipPolicy` in Spring Batch to skip invalid records without interrupting the job:

```
java

@Bean
public SkipPolicy skipPolicy() {
```

```
        return (throwable, skipCount) -> {
            return skipCount < 10 && throwable
    instanceof ValidationException;
        };
     }
```

4. **Testing**: Simulate migration in a test environment with smaller datasets.

This setup ensures efficient batch processing and robust validation, providing consistency and reliability for your Oracle database migration project.

Copy Ask Perplexity