Reactive Programming: A New Asynchronous Database Access API

Rajesh Tiwary, JDBC Developer, Oracle
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Program Agenda

1. Overview
   - Basic Concepts
   - Code and more concepts
   - Wrap-up
Asynchronous Java Database Access

• Java standard database access API that never blocks user threads
• Developed by the JDBC Expert Group with community input
• Targeted for a near future release of JAVA
• Asynchronous apps have better throughput
  – Fewer threads means less thread scheduling, less thread contention
  – Database access is slow so blocked threads leave resources idle for a long time
  – Simultaneous access to multiple databases, e.g., sharded databases, map/reduce
  – Fire and forget, e.g., DML, stored procedures
Goals

• No user thread blocks
  – Minimize the number of threads used for database access
• Alternate API for database access
  – Not an extension to the current JDBC API
  – Not a replacement for the current JDBC API
• Target high throughput apps
  – Not a completely general purpose database access API
  – The initial version will have a limited feature set
• Build on the Java SE class library
Design Choices

• Minimal or no reference to java.sql
• Rigorous use of types
• Builder pattern
• Fluent API
• Immutable after initialization
• One way to do something is enough
• Avoid SQL processing
• Avoid callback hell
What About …?

There are already multiple async Java and JavaScript APIs

- Streams
  - Java streams are inherently synchronous
- Reactive Streams
- NodeJS
- ADBCJ
Program Agenda

1. Async JDBC API
2. Basic Concepts
   - Code and more concepts
3. Wrap-up
CompletionStage interface & CompletableFuture class

• Java class library mechanism for asynchronous style programming
  – Brings reactive programming to Java
  – Enables the composition of asynchronous tasks.
  – A task has a state; it might be:
    • running
    • completed normally with a result
    • completed exceptionally with an exception
  – Event based: the result of the completion is pushed to dependent tasks
    • push model -> higher scalability than pull or poll
• Supports lambda expressions and fluent programming
• More @ http://bit.ly/2nnLqa0
Execution Model

• Everything is an Operation = a result + a CompletionStage
  – SQL or other database operation
  – Parameter assignments
  – Result handling
  – Submission and CompletableFuture

• User thread creates and submits Operations
  – Creating and submitting never blocks; user thread never blocks

• Implementation executes those Operations asynchronously
  – Performs round trip(s) to the database
  – Executes result handling
  – Completes CompletableFuture
Program Agenda

1. Async JDBC API
2. Basic concepts
3. Code and more concepts
4. Wrap-up
public void trivialInsert(DataSource ds) {
    String sql = "insert into tab values (:id, :name, :answer)";
    try (Connection conn = ds.getConnection()) {
        conn.countOperation(sql)
            .set("id", 1, JdbcType.NUMERIC)
            .set("name", "Deep Thought", JdbcType.VARCHAR)
            .set("answer", 42, JdbcType.NUMERIC)
            .submit();
    }
}
All SQL is Vendor Specific

- No escape sequences
- No specified parameter markers
- Non vendor specific syntax requires processing by the driver
  - Adds overhead
  - Increases code complexity
  - Minimal benefit as most apps are tied to a specific database regardless

Note: Code examples use parameter markers from a variety of databases including DB2 (:foo), MySQL (?), Oracle Database(:foo), PostgresSQL($1), and SQL Server (@foo).
Trivial Select

```java
public void trivialSelect(DataSource ds, List<Integer> result) {
    String sql = "select id, name, answer " + "from tab where answer = $1";
    try (Connection conn = ds.getConnection()) {
        conn.<List<Integer>>rowOperation(sql)
            .set("1", 42, JdbcType.NUMERIC)
            .rowAggregator((ignore, row) -> {
                result.add(row.get("id", Integer.class));
                return null;
            })
            .submit();
    }
}
```
Connection Properties

```java
public DataSource getDataSource(String url, String user, String pass) {
    return DataSourceFactory.forVendor("Oracle Database")
        .builder()
        .url("jdbc:oracle:nonblocking:@//javaone.oracle.com:5521/javaone")
        .username("scott")
        .password("tiger")
        .connectionProperty(JdbcConnectionProperty.TRANSACTION_ISOLATION, TransactionIsolation.SERIALIZABLE)
        .connectionProperty(NLS_LANGUAGE, "French")
        .build();
}
```
Getting a Connection

In interface DataSource:

```java
public default Connection getConnection() {
    return builder().build().connect();
}
```

Connection:

```java
public default Connection connect() {
    holdForMoreMembers();
    submit();
    connectOperation().submit();
    return this;
}
```
Connection Pooling

- *in interface Connection*
- `public Connection activate();`
- `public Connection deactivate();`
- `public registerLifecycleListener(LifecycleListener listener);`
Basic SELECT

```java
public Future<Integer> selectIdForAnswer(DataSource ds, int answer) {
    String sql = "select id, name, answer from tab " + "where answer = @target";
    try (Connection conn = ds.getConnection()) {
        return conn.<List<Integer>>rowOperation(sql)
            .set("target", answer, JdbcType.NUMERIC)
            .initialValue((List) -> new ArrayList<>())
            .rowAggregator((list, row) -> {
                list.add(row.get("id", Integer.class));
                return list;
            })
            .submit()
            .toCompletableFuture()
            .thenApply(l -> l.get(0));
    }
}
```
public static class Item {
    protected int id;
    protected String name;
    protected int answer;
    @SqlColumns({"ID","USER","RESPONSE"})
    public Item(int id, String name, int answer) {
        this.id = id;
        this.name = name;
        this.answer = answer;
    }
}

(cont.)
POJOs (cont.)

```java
@SqlParameter("id", "NUMERIC")
public int getId() {
    return id;
}

@SqlParameter("name", "VARCHAR")
public String getName() {
    return name;
}

@SqlParameter("answer", "NUMERIC")
public int getAnswer() {
    return answer;
}
```
OperationGroup

Operations can be grouped
• OperationGroup has its own result handling and CompletableFuture
• Members submitted to group. OperationGroup submitted as a unit
• Order of execution
  – Sequential in order submitted
  – Parallel, any order
• Error response
  – Dependent: remaining group members skipped
  – Independent: remaining group members unaffected
• Conditional or unconditional
• Connection is an OperationGroup
  – Sequential, dependent, unconditional by default
Batch INSERT

public void insertListIndependent(List<Item> list, DataSource ds) {
  String sql = "insert into tab values " + 
    "(:elem_id, :elem_name, :elem_answer)";
  try (Connection conn = ds.getConnection()) {
    BatchCountOperation batch = conn.batchCountOperation(sql);
    list.forEach( (elem) -> {
      batch.countOperation()  
        .set("elem_", elem)  
        .submit();
    });
    batch.submit();
  }
}
public void insertListHold(List<Item> list, DataSource ds) {
    String sql = "insert into tab " +
        "values (@elem_id, @elem_name, @elem_answer)";
    try (Connection conn = ds.getConnection()) {
        OperationGroup group = conn.operationGroup()
            .holdForMoreMembers()
            .independent();
        group.submit();
    }

    (cont.)
Independent INSERT (cont.)

```java
for (Item elem : list) {
    group.countOperation(sql)
        .set("elem_", elem)
        .submit()
        .toCompletableFuture()
        .exceptionally( t -> {
            System.out.println(elem.getId());
            return null;
        });
}

group.releaseProhibitingMoreMembers();
```
Future Parameters

```java
public void deposit(Connection conn, int accountId, double amount) {
    String selectSql = "SELECT balance FROM account WHERE id = $1";
    CompletableFuture<Double> newBalanceF = conn.<Double>rowOperation(selectSql)
        .set("1", accountId, JdbcType.INTEGER)
        .rowAggregator((p, row) -> row.get("balance", Double.class))
        .submit()
        .thenApply(b -> b + amount);
    String updateSql = "UPDATE account SET balance=$2 WHERE id = $1";
    conn.countOperation(updateSql)
        .set("1", accountId, JdbcType.INTEGER)
        .set("2", newBalanceF, JdbcType.DOUBLE)
        .submit();
}
```
public void updateListParallel(List<Item> list, DataSource ds) {
    String query = "select id from tab where answer = ?";
    String update = "update tab set answer = ? where id = ?";
    try (Connection conn = ds.getConnection()) {
        OperationGroup<Object, Object> group = conn.operationGroup()
            .holdForMoreMembers()
            .independent()
            .parallel();
        group.submit();
        for (Item elem : list) {
            CompletableFuture<Integer> idF = group.<Integer>rowOperation(query)
                .set("1", elem.getAnswer(), JdbcType.NUMERIC)
                .rowAggregator( (Integer ignore, Result.Row row) ->
                    row.get("id", Integer.class) )
                .submit();
        }
    } (cont.)
Parallel UPDATE (cont.)

group.countOperation(update)
  .set("1", idF)
  .set("2", "42")
  .submit()
  .toCompletableFuture()
  .exceptionally( t -> {
    System.out.println("Update failed: " + elem.getId());
    return null;
  });
}

  group.releaseProhibitingMoreMembers();
}
Close

public default void close() {
    closeOperation().submit();
    releaseProhibitingMoreMembers();
}

Note: A CloseOperation is never skipped.
Program Agenda

1. Async JDBC API
2. Basic Concepts
3. Code and more concepts
4. Wrap-up
Status

• Everything subject to change
• ADBA APIs are Developed by the JDBC Expert Group through the Java Community Process
• Targeted for a near future release of JAVA
• Available for download from OpenJDK at
  • http://oracle.com/goto/java-async-db
• AoJ(ADBA over JDBC) implements much of the API
  • https://github.com/oracle/oracle-db-examples/tree/master/java/AoJ
• Send feedback to jdbc-spec-discuss@openjdk.java.net
Integrated Cloud
Applications & Platform Services
GREAT INDIAN DEVELOPER SUMMIT 2019
Conference: April 23-26, Bangalore

Register early and get the best discounts!

www.developersummit.com
@greatindiandev
bit.ly/gidslinkedin
facebook.com/gids19
bit.ly/saltmarchyoutube
flickr.com/photos/saltmarch/