ORACLE®

JavaOne™

ORACLE®

JDBC Next: A New Asynchronous API for Connecting to a Database

CON1491

Douglas Surber
JDBC Architect
Oracle Database Server Technologies

October 3, 2017

JavaYourNext
(Cloud)

Safe Harbor Statement

Program Agenda

1. Overview
2. Code
3. Wrap-up
4. Q&A

Overview

JavaYourNext (Cloud)

Asynchronous 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, Java 10 or equivalent
- 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. map/reduce, sharded databases
  - Fire and forget, e.g. DMS, stored procedures

Goals

- No user thread blocks ever
  - Minimize the number of threads used for database access
- Alternate API for database access
  - Not an extension of the standard JDBC API
  - Not a replacement for the standard 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 asynchronous/non-blocking database access APIs

- Streams
  - Java streams are inherently synchronous
- ReactiveStreams
  - Not integrated into the Java SE class library
- NodeJS
- ADBCJ

Code

JavaYourNext
(Cloud)

Trivial Insert

```
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

```
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();
  }
}
```

Execution Model

Everything is an Operation

- Operations consist of
  - 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 CompletableFutures

Getting a DataSource

```
public DataSource getDataSource(String url, String user, String pass) {
  return DataSourceFactory.forName("Oracle Database")
          .builder()
          .url("oracle:database:@//javaone.oracle.com:5521/javaone")
          .username("scott")
          .password("tiger")
          .connectionProperty(JdbcConnectionProperty.TRANSACTION_ISOLATION,
                              TransactionIsolation.SERIALIZABLE)
          .build();
}
```

Getting a Connection

in interface DataSource:

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

in interface Connection:

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

Basic SELECT

```
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", 42, JdbcType.NUMERIC)
            .initialValue( () -> new ArrayList<>() )
            .rowAggregator( (list, row) -> {
                  list.add(row.get("id", Integer.class));
                  return list;
            } )
            .submit()
            .toCompletableFuture()
            .thenApply( l -> l.get(0) );
  }
}
```

POJOs

```
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.)*

```
  @SqlParameter(marker=”id”, sqlType="NUMERIC")
  public int getId() {
    return id;
  }

  @SqlParameter(marker=”name”, sqlType="VARCHAR")
  public String getName() {
    return name;
  }

  @SqlParameter(marker=”answer”, sqlType="NUMERIC")
  public int getAnswer() {
    return answer;
  }
}
```

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();
    }
}
```

OperationGroup

Operations can be grouped

- OperationGroup has its own result handling and CompletableFuture
- Members submitted to group. OperationGroup is 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

Independent INSERT

```
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.*)

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

Close

in interface Connection

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

Note: A `CloseOperation` is never skipped.

Connection Pooling

in interface Connection

```
public Connection activate();

public Connection deactivate();

public registerLifecycleListener(LifecycleListener listener);
```

ConnectionProperties

```
public DataSource getDataSource(String url, String user, String pass) {
  return DataSourceFactory.forName("Oracle Database")
          .builder()
          .url("oracle:database:@//javaone.oracle.com:5521/javaone")
          .username("scott")
          .password("tiger")
          .connectionProperty(JdbcConnectionProperty.TRANSACTION_ISOLATION,
                              TransactionIsolation.SERIALIZABLE)
          .connectionProperty(NLS_LANGUAGE, “French”)
          .build();
}
```

ConnectionProperty

```
public static class NlsLanguageProperty implements ConnectionProperty {
  public static final ConnectionProperty NLS_LANGUAGE
      = new NlsLanguageProperty();
  private NlsLanguageProperty() {}
  public String name() { return “NLS_LANGUAGE”; }
  public Class range() { return String.class; }
  public String defaultValue() { return “American”; }
  public boolean isSensitive() { return false; }
  public Operation configureOperation(OperationGroup group, Object value) {
    String sql = “ALTER SESSION SET NLS_LANGUAGE TO “ + value;
    return group.operation(sql);
  }
```

ARRAYs and STRUCTs

```
@SqlArray(elementSqlTypeName=”STUDENT”)
public class Roster extends LinkedList<Student> {
  public Roster() {};
}

@SqlStruct(sqlTypeName=”STUDENT”, fields={
  @Field(sqlFieldName=”NAME”, sqlTypeName=”VARCHAR(100)”, javaFieldName=”name”),
  @Field(sqlFieldName=”EMAIL”, sqlTypeName=”VARCHAR(100)”,
javaFieldName=”email”)})
public class Student {
  String name;
  String email;
  public void setName(String n) {name = n;}
  public void setEmail(String e) {email = e;}
  public String getName() {return name;}
  public String getEmail() {return email;}
}
```

Transactions

```
public void transfer(Connection conn,
                     double amount, int fromAccount, int toAccount) {
  String sql = “update account set balance=balance+@amount where id = @account”;
  conn.countOperation(sql)
      .set(“amount”, -amount, JdbcType.DECIMAL)
      .set(“account”, fromAccount, JdbcType.INTEGER)
      .submit();
  conn.countOperation(sql)
      .set(“amount”, amount, JdbcType.DECIMAL)
      .set(“account”, toAccount, JdbcType.INTEGER)
      .submit();
  conn.commit();
}
```

Transactions *(cont.)*

```
public void transfer(Connection conn,
                     double amount, int fromAccount, int toAccount) {
  String sql = “update account set balance=balance+@amount where id = @account”;
  final Transaction tran = conn.getTransaction();
  conn.countOperation(sql)
      .set(“amount”, -amount, JdbcType.DECIMAL)
      .set(“account”, fromAccount, JdbcType.INTEGER)
      .onError( e -> tran.setRollbackOnly() )
      .submit();
  conn.countOperation(sql)
      .set(“amount”, amount, JdbcType.DECIMAL)
      .set(“account”, toAccount, JdbcType.INTEGER)
      .onError( e -> tran.setRollbackOnly() )
      .submit();
  conn.commit();
}
```

Local Operations

```
public void paycheck(Connection conn, Employee emp) {
  String sql = “call generate_paycheck(?, ?)”;
  CompletableFuture<CheckDetail> details = conn.<CheckDetail>outOperation(sql)
      .set(“1”, emp, JdbcType.STRUCT)
      .outParameter(“2”, JdbcType.STRUCT)
      .resultProcessor( m -> m.get(“2”, CheckDetail.class) )
      .submit()
      .toCompletableFuture();
  conn.localOperation()
      .onExecution( () -> { printPaycheck(details); return null; } )
      .submit()
      .toCompletableFuture()
      .thenRun( () -> reportPrintSuccess(details) );
      .exceptionally( t -> {reportPrintFailure(details); return null;} );
}
```

Future Parameters

```
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()
      .toCompletableFuture()
      .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();
}
```

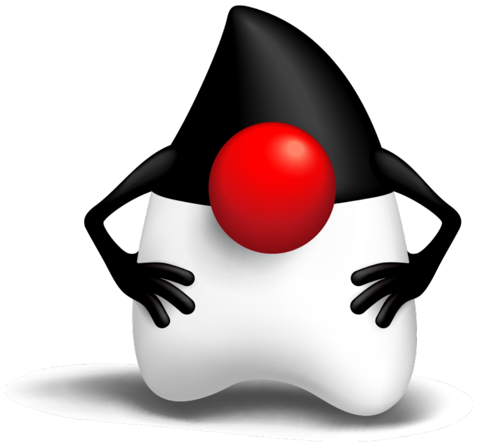
Wrap-up

JavaYourNext

(Cloud)

Status

- **Everything is subject to change**
- Prototype Oracle Database driver implements much of the API
  - Uses nio Selector so non-blocking all the way down
- Developed by the JDBC Expert Group through the Java Community Process
- Targeted for a near future release, Java 10 or equivalent
- **The API is available for download from OpenJDK at http://oracle.com/goto/java-async-db**
- Send feedback to jdbc-spec-discuss@openjdk.java.net



Q&A

JavaYourNext (Cloud)

Integrated Cloud

Applications & Platform Services

JavaOne™

ORACLE®

ORACLE®

```java
/*
 * Copyright (c)  2017, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */
package test;

import java.sql2.BatchCountOperation;
import java.sql2.Connection;
import java.sql2.ConnectionProperty;
import java.sql2.DataSource;
import java.sql2.DataSourceFactory;
import java.sql2.JdbcConnectionProperty;
import java.sql2.JdbcConnectionProperty.TransactionIsolation;
import java.sql2.JdbcType;
import java.sql2.Operation;
import java.sql2.OperationGroup;
import java.sql2.SqlArray;
import java.sql2.SqlColumns;
import java.sql2.SqlParameter;
import java.sql2.SqlStruct;
import java.sql2.SqlStruct.Field;
import java.sql2.Transaction;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.Future;
import java.util.function.Consumer;

/**
 *
 * Code from JavaOne 2017 slides: CON-1491
 * JDBC Next: A New Asynchronous API for Connecting to a Database
 *
 */
public class JavaOne {

  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();
    }
  }

 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();
    }
  }

  public DataSource getDataSource(String url, String user, String pass) {
    return DataSourceFactory.forName("Oracle Database")
            .builder()
            .url("oracle:database:@//javaone.oracle.com:5521/javaone")
            .username("scott")
            .password("tiger")
            .connectionProperty(JdbcConnectionProperty.TRANSACTION_ISOLATION,
                    TransactionIsolation.SERIALIZABLE)
            .build();
  }

  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", 42, JdbcType.NUMERIC)
              .initialValue(() -> new ArrayList<>())
              .rowAggregator((list, row) -> {
                list.add(row.get("id", Integer.class));
                return list;
              })
              .submit()
              .toCompletableFuture()
              .thenApply(l -> l.get(0));
    }
  }

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

  public DataSource getDataSource_2(String url, String user, String pass) {
    return DataSourceFactory.forName("Oracle Database")
            .builder()
            .url("oracle:database:@//javaone.oracle.com:5521/javaone")
            .username("scott")
            .password("tiger")
            .connectionProperty(JdbcConnectionProperty.TRANSACTION_ISOLATION,
                    TransactionIsolation.SERIALIZABLE)
            .connectionProperty(NlsLanguageProperty.NLS_LANGUAGE, "French")
            .build();
  }

  public void transfer(Connection conn,
          double amount, int fromAccount, int toAccount) {
    String sql = "update account set balance=balance+@amount where id =
@account";
    conn.countOperation(sql)
            .set("amount", -amount, JdbcType.DECIMAL)
            .set("account", fromAccount, JdbcType.INTEGER)
            .submit();
    conn.countOperation(sql)
            .set("amount", amount, JdbcType.DECIMAL)
            .set("account", toAccount, JdbcType.INTEGER)
            .submit();
    conn.commit();
  }

  public void transfer_2(Connection conn,
          double amount, int fromAccount, int toAccount) {
    String sql = "update account set balance=balance+@amount where id =
@account";
    final Transaction tran = conn.getTransaction();
    conn.countOperation(sql)
            .set("amount", -amount, JdbcType.DECIMAL)
            .set("account", fromAccount, JdbcType.INTEGER)
            .onError(e -> tran.setRollbackOnly())
            .submit();
    conn.countOperation(sql)
            .set("amount", amount, JdbcType.DECIMAL)
            .set("account", toAccount, JdbcType.INTEGER)
            .onError(e -> tran.setRollbackOnly())
            .submit();
    conn.commit();
  }

  public void paycheck(Connection conn, Employee emp) {
    String sql = "call generate_paycheck(?, ?)";
    CompletableFuture<CheckDetail> details =
conn.<CheckDetail>outOperation(sql)
            .set("1", emp, JdbcType.STRUCT)
            .outParameter("2", JdbcType.STRUCT)
            .resultProcessor(m -> m.get("2", CheckDetail.class))
            .submit()
            .toCompletableFuture();
    conn.localOperation()
            .onExecution(() -> {
              printPaycheck(details);
              return null;
            })
            .submit()
            .toCompletableFuture()
            .thenRun(() -> reportPrintSuccess(details))
            .exceptionally(t -> {
              reportPrintFailure(details);
              return null;
            });
  }
  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()
            .toCompletableFuture()
            .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 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;
    }

    @SqlParameter(marker = "id", sqlType = "NUMERIC")
    public int getId() {
      return id;
    }

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

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

```
public static class NlsLanguageProperty implements ConnectionProperty {

    public static final ConnectionProperty NLS_LANGUAGE
            = new NlsLanguageProperty();

    private NlsLanguageProperty() {
    }

    @Override
    public String name() {
      return "NLS_LANGUAGE";
    }

    @Override
    public Class range() {
      return String.class;
    }

    @Override
    public String defaultValue() {
      return "American";
    }

    @Override
    public boolean isSensitive() {
      return false;
    }

    @Override
    public Operation configureOperation(OperationGroup group, Object value) {
      String sql = "ALTER SESSION SET NLS_LANGUAGE TO " + value;
      return group.operation(sql);
    }
  }
```

```
  @SqlArray(elementSqlTypeName = "STUDENT")
  public static class Roster extends LinkedList<Student> {

    public Roster() {
    }

  }

@SqlStruct(sqlTypeName = "STUDENT", fields = {
    @Field(sqlFieldName = "NAME", sqlTypeName = "VARCHAR(100)", javaFieldName = "name"),
    @Field(sqlFieldName = "EMAIL", sqlTypeName = "VARCHAR(100)", javaFieldName = "email")})
  public class Student {

    String name;
    String email;

    public void setName(String n) {
      name = n;
    }

    public void setEmail(String e) {
      email = e;
    }

    public String getName() {
      return name;
    }

    public String getEmail() {
      return email;
    }
  }

  public class Employee {
  }

  public class CheckDetail {
  }

  private void printPaycheck(Future<CheckDetail> checkF) {
    throw new UnsupportedOperationException("Not supported yet."); //To change body of
generated methods, choose Tools | Templates.
  }

  private Consumer<? super Object> reportPrintSuccess(Future<CheckDetail> checkF) {
    throw new UnsupportedOperationException("Not supported yet."); //To change body of
generated methods, choose Tools | Templates.
  }

  private CompletableFuture<Void> reportPrintFailure(Future<CheckDetail> checkF) {
    throw new UnsupportedOperationException("Not supported yet."); //To change body of
generated methods, choose Tools | Templates.
  }

}
```