MySQL Connector/Node.js and the X DevAPI

Live for the Code

Rui Quelhas
Software Developer
Oracle, MySQL
October 25, 2018

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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

whoami

Rui Quelhas

MySQL Middleware and Clients team (Connectors)

Connector/Node.js Lead Developer

rui.quelhas@oracle.com

@ruiquelhas

Outline

1. Building blocks for modern MySQL
2. Diving into the X DevAPI
3. X DevAPI via Connector/Node.js
4. Final thoughts

Building blocks for modern MySQL

X DevAPI

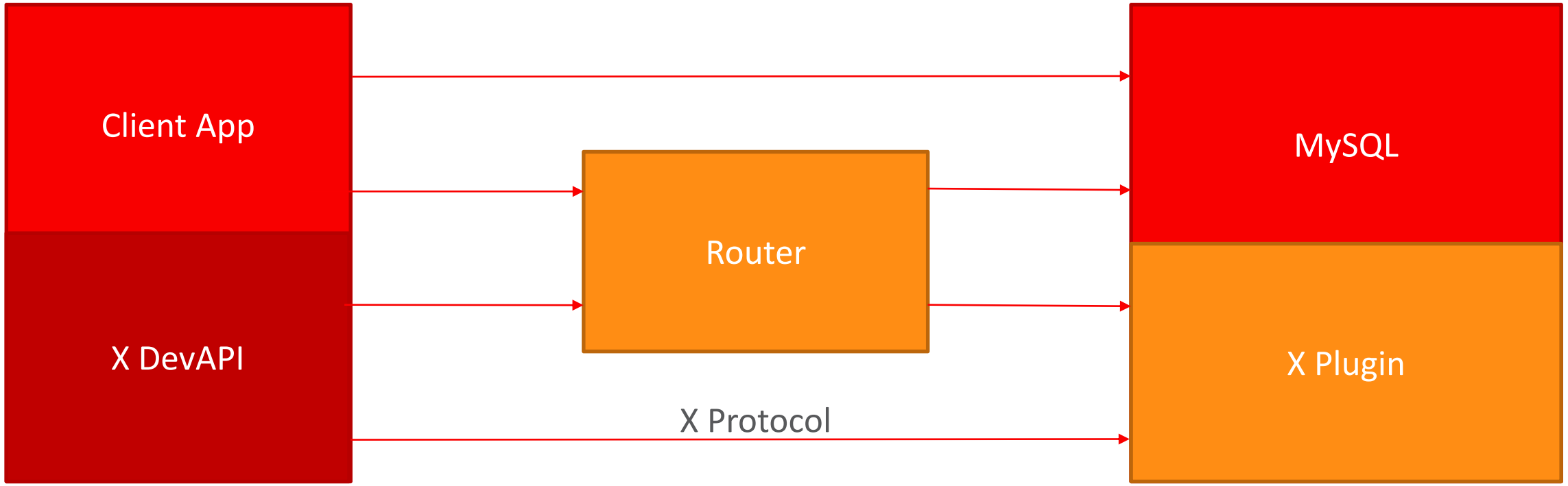
- High-level database API to develop modern applications powered by **InnoDB Cluster** [TIP3065, HOL2988]
- Application gateway for the underlying document-store infrastructure (X Protocol, Router, X Plugin)
- Off-the-shelf support for **CRUD NoSQL** document operations
- Lower-level management via raw SQL and other advanced features
- Standard specification for all the official client implementations
- Available in connectors for the most popular languages as well as the brand new MySQL Shell

MySQL Document Store [DEV5986]

- It’s now possible to save and retrieve unstructured data using a MySQL database (not talking about raw JSON)
- Data lives in a JSON column but everything is abstracted way from the user, which only deals with documents
- No need to worry about schemas and data types
- Keeping logical consistency and ACID (it’s MySQL after all)
- At the same time, a schema can still mix in traditional tables
- ...and more https://lefred.be/content/top-10-reasons-for-nosql-with-mysql/

ORACLE

Component Overview



https://dev.mysql.com/doc/refman/8.0/en/document-store.html

Router

- MySQL Router is an integral part of **InnoDB Cluster**
- Lightweight middleware that provides transparent routing between an application and back-end MySQL servers
- Can be used for a wide variety of use cases, but mainly to address high availability and scalability concerns
- Runs as a standalone component
- Packaged with the MySQL server for the usual platforms
- Documentation available at https://dev.mysql.com/doc/mysql-router/8.0/en/

X Plugin

- New MySQL server plugin (developed internally)
- Installed and enabled by default on MySQL 8.0 series
- Provides the document store infrastructure on top of plain MySQL servers
- Uses a different set of ports (**33060** instead of the usual MySQL **3306**)
- Implements a client-server protocol based on Google Protocol Buffers
- Accepts X DevAPI client requests and processes them using the core MySQL engine
- Documentation available at https://dev.mysql.com/doc/refman/8.0/en/document-store.html

X Protocol

- New client-server protocol based on Google Protocol Buffers (easier for **ANYONE** to extend and improve)
- Content awereness
  - CRUD expression trees to establish operation boundaries
  - Expectations to define conditions and create pipelined statements
  - Placeholder assignment and value escaping
- Security baked in
  - SSL/TLS by default
  - No information leaked to unauthenticated users (e.g. server version)
- Open source at https://dev.mysql.com/doc/internals/en/x-protocol.html

X Protocol

Message example

```
Mysqlx.Crud.Find {
  collection { name: "collection_name", schema: "schema_name" }
  data_model: DOCUMENT
  criteria {
    type: OPERATOR
    operator {
      name: "=="
      param {
        type: IDENT,
        identifier { name: "_id" }
      }
      param {
        type: LITERAL,
        literal {
          type: V_STRING,
          v_string: { value: "some_string" }
        }
      }
    }
  }
}
```

Building upon the X Protocol

- Each client offers a unified (up to some point) flavour of the X DevAPI to be consumed by an application
- X DevAPI constructs are exposed to the user via contextual methods operating on different database objects through the hierarchy
- Database interactions are encoded as X Protocol messages expressing the respective action
- Level of abstraction varies depending on the task
- Represents the vision of the MySQL team
- However, anyone can create their own API flavour on top of the protocol

Diving into the X DevAPI

Overview

- Wraps powerful concepts in a simple API
  - High-level session concept for writing code that transparently scales from single MySQL Server to a multi-server environment
  - Operations are simple, expressive and easy to understand
  - Non-blocking, asynchronous calls follow common host language patterns.
- Introduces a new, modern and easy-to-learn way to work with data
  - Documents are stored in Collections and have their dedicated CRUD operation set
  - Work with your existing domain objects or generate code based on structure definitions for strictly typed languages
  - Focus is put on working with data via CRUD operations (even for regular tables)
  - Semantic query-building preferred over traditional SQL string mashing

What do I get?

- Fluent API and flexible parameters
- Multiple connection modes (with support for connection pooling)
- Automatic connection failover and timeout management
- Security best practices with multiple SSL/TLS operation modes and authentication mechanisms as well as improved SQL injection protection
- Small and portable expression-based query language (subset of SQL)
- Hybrid CRUD API for both Document store collections and regular tables
- SQL statements for everything else
- ACID (transactions and row locking)

Fluent API

- Code flows from a single point of entry – ***getSession()***
- Code becomes more readable, maintainable (and even testable)
- Operations encapsulated in specialized and semantic methods
- Nice scaffolding for refactor tasks
- First-class support for text-editor (or IDE) hints and auto-completion
- Smaller SQL injection surface area
- Common standard between different programming environments

Flexible Parameters

- Most public API methods work with:
  - Multiple individual parameters
  - A single array of parameters
  - An object with named properties (where it applies)

```
mysqlx.getSession('root@localhost')
mysqlx.getSession({ user: 'root' })

collection.add({ name: 'foo' }).add({ name: 'bar' })
collection.add([{ name: 'foo' }, { name: 'bar' }])

collection.find('name = :name').bind('name', 'foo')
collection.find('name = :name').bind({ name: 'foo' })
collection.find().fields('foo', 'bar')
collection.find().fields(['foo', 'bar'])
```

Connection Types

Different operation modes

- Each X DevAPI **Session** matches a single connection (regular mode)
- Pooling mode (X DevAPI **Session**s create or re-use existing connections)

```
const url = 'mysqlx://root@localhost:33060/defaultSchema'

// create a regular standalone connection
mysqlx.getSession(url)

const pooling = { enabled: true, maxIdleTime: 500, maxSize: 25, queueTimeout: 500 }

const client = mysqlx.getClient(url, { pooling })

// acquire a connection from the pool (creates one if it does not exist)
client.getSession()
```

Connection failover and timeout

Pooling and non pooling setups

- Automatic priority-based failover
- Different timeout management for single-host or multi-host settings

```
// single-host
mysqlx.getSession('mysqlx://root@localhost:33060?connect-timeout=1000');


// multi-host

// order-based priority
mysqlx.getSession('mysqlx://root@[localhost:33060, localhost:33061]?connect-timeout=1000')
mysqlx.getSession('root@[localhost:33060, localhost:33061] ')

// weight-based priority
mysqlx.getSession('mysqlx://root@[(localhost:33061, priority=99), (localhost:33060, priority=100)]')
mysqlx.getSession('root@[(localhost:33061, priority=99), (localhost:33060, priority=100)]?connect-timeout=1000')
```

Security Options

- SSL/TLS is enabled by default, for TCP connections, when creating a session
- Local UNIX socket connections disable it (not needed) to be faster
- Additional server certificate validation
- Options can be overridden on-demand

```
mysqlx.getSession({ ssl: false })
mysqlx.getSession('mysqlx://root@localhost?ssl-mode=DISABLED')

mysqlx.getSession({ ssl: true, sslOptions: { ca: '/path/to/ca.pem' } })
mysqlx.getSession('mysqlx://root@localhost?ssl-ca=(/path/to/ca.pem)')

mysqlx.getSession({ ssl: true, sslOptions: { ca: '/path/to/ca.pem', crl: '/path/to/crl.pem' } })
mysqlx.getSession('mysqlx://root@localhost?ssl-ca=(/path/to/ca.pem)&ssl-crl=(/path/to/crl.pem)')
```

Authentication Mechanisms

- SHA-1 and SHA-2 password hashing
- Supports authentication mechanisms for multiple server plugins:
  - **MYSQL41** for **mysql_native_password**
  - **PLAIN** and **SHA256_MEMORY** for **sha256_password** and/or **caching_sha2_password**

```
mysqlx.getSession({ user: 'root', auth: 'MYSQL41' })
mysqlx.getSession('mysqlx://root@localhost?auth=MYSQL41')

mysqlx.getSession({ user: 'root', auth: 'PLAIN' })
mysqlx.getSession('mysqlx://root@localhost?auth=PLAIN')

mysqlx.getSession({ user: 'root', auth: 'SHA256_MEMORY' })
mysqlx.getSession('mysqlx://root@localhost?auth=SHA256_MEMORY')
```

Hybrid CRUD API

Table	Collection
• `insert()` • `select()` • `dpdate()` • `delete()`	• `add()` • `find()` • `modify()` • `remove()`

- Operational methods return proper database object or operation instances
- CRUD operations: **CollectionAdd**, **TableInsert**, **CollectionFind**, **TableSelect**, **CollectionModify**, **TableUpdate**, **CollectionRemove**, **TableDelete**
- Operation boundaries - filtering criteria, computed projections, placeholder assignment, resultset ordering - established using an expression language

Expression Strings

- Small subset of the SQL language
- Easy to grasp (everyone already knows SQL) but still powerful
- Expressive and human-readable
- Common standard between all the official connector implementations

```
// JavaScript
collection
  .find("name = 'foo' AND age > 42")
  .fields("name", "age")
  .groupBy("name", "age")
  .sort("name ASC", "age DESC")
  .limit(4)
  .offset(2)
  .execute()
```

```
// Java
collection
  .find("name = 'foo' AND age > 42")
  .fields("name", "age")
  .groupBy("name", "age")
  .sort("name ASC", "age DESC")
  .limit(4)
  .offset(2)
  .execute()
```

Placeholder assignment

- Placeholders are supported by the expression grammar
- Variable values are native to the protocol itself
- Additional server-side validation and escaping
- Mitigates SQL injection risks

```
collection.find('name = :name')
  .bind('name', 'foo')
  .execute()

collection.find('name = :name')
  .bind({ name: 'foo' })
  .execute()
```

```
Mysqlx.Crud.Find {
  criteria {
    operator {
      param {
        identifier { document_path: "name" }
      }
      param { position: 0 }
    }
  }
  args { v_string: "foo" }
}
```

Raw SQL

- For ETL, structured analytics and/or reporting, SQL is always an option
- Allows to tap into features not yet part of the X DevAPI, such as:
  - DDL operations for relational tables
  - Managing key constraints
  - Using JOINs

```
// create a table
session.sql('CREATE TABLE foo (bar VARCHAR(3))').execute()

// add a unique constraint
session.sql('ALTER TABLE foo ADD COLUMN bar VARCHAR(3) GENERATED ALWAYS AS doc->>"$.bar" VIRTUAL UNIQUE KEY NOT
NULL').execute()

// execute a JOIN query
session.sql('SELECT DISTINCT t1.bar FROM foo t1 JOIN baz t2 ON t1.bar = t2.qux WHERE t1.qux = t2.quux').execute()
```

Transactions and Savepoints

- Session-level atomic operations
- Create, commit or rollback transactions in the scope of a session
- Create, release or rollback to intermediate savepoints in those transactions

```
try {
  session.startTransaction()
  // run some operations (1)
  session.createSavepoint('foo')
  // run more operations (2)
  session.releaseSavepoint('foo')
  session.commit()
} catch (err) {
  try {
    session.rollbackTo('foo') // go to (2)
  } catch (err) {
    session.rollback() // revert the entire thing
  }
}
```

Row Locking

- Reads determine isolation level in the presence of concurrent transactions
- **Exclusive** and **Shared** locks with different operation modes:
  - **DEFAULT** (regular `SELECT FOR ... UPDATE` or `SELECT FOR ... SHARE` behavior)
  - **NOWAIT** (query executes immediately and fails if row is locked)
  - **SKIP_LOCKED** (query executes immediately and ignores row)

```
// works for table.select() as well
collection.find('name = "foo"').lockExclusive() // mysqlx.LockContention.DEFAULT
collection.find('name = "foo"').lockExclusive(mysqlx.LockContention.NOWAIT)
collection.find('name = "foo"').lockExclusive(mysqlx.LockContention.SKIP_LOCKED)

// works for table.select() as well
collection.find('name = "bar"').lockShared() // mysqlx.LockContention.DEFAULT
collection.find('name = "bar"').lockShared(mysqlx.LockContention.NOWAIT)
collection.find('name = "bar"').lockShared(mysqlx.LockContention.SKIP_LOCKED)
```

X DevAPI via Connector/Node.js

Overview

- Currently the only Node.js driver for MySQL with out-of-the-box support for 8.0 servers and, particularly, the X DevAPI
- Open source, available on GitHub and on npm
- Modern Node.js asynchronous API using ***Promise***s
- Covers a big chunk of the X DevAPI spec:
  - Fluent API and query builder based on the common expression language
  - Document store and relational CRUD methods alongside plain SQL support
  - Secure by default (SSL/TLS and SHA256 authentication)
  - Connection pooling and failover
  - Logical consistency with transactions, savepoints and row-locking...

Asynchronous API

- Operations are sent to the server when the ***execute()*** method is called
  - receives an optional query operation callback push-based cursor) which runs for each element in the result set
  - returns a ***Promise*** which resolves to a ***Result*** instance containing details and metadata about the operation, or fails with an ***Error*** (can be used with ***async/await)***

```
collection
  .find()
  .execute(doc => {
    // do something with the document
  })
  .then(result => {
    // result contains details about the operation
  })
  .catch(err => {
    // handle any errors
  })
```

```
collection
  .remove()
  .execute(/* no cursor will be needed */)
  .then(result => {
    // result contains details about the operation
  })
  .catch(err => {
    // handle any errors
  })
```

Up and running

```
$ npm install --save --save-exact @mysql/xdevapi
```

```
const mysqlx = require('@mysql/xdevapi');
```

```
(async function () {

  const session = await mysqlx.getSession({ user: 'root' });
  const schema = session.getSchema('codeone2018');

  if (!(await schema.existsInDatabase())) {
    await session.createSchema('codeone2018');
  }

  // ...

})();
```

Data definition

```
// const schema = session.getSchema('codeone2018');
await schema.createCollection('sessions_nosql', { ReuseExistingObject: true });


// For now, table DDL operations are performed using SQL only
await session.sql('USE codeone2018').execute();
await session.sql(`CREATE TABLE sessions_sql (
  _id VARCHAR(7) NOT NULL,
  title VARCHAR(250),
  PRIMARY KEY (_id)
)`).execute();


const collections = await schema.getCollections();
console.log(collections); // [{ schema: 'codeone2018', collection: 'sessions_nosql' }]

const tables = await schema.getTables();
console.log(tables); // [{ schema: 'codeone2018', table: 'sessions_sql' }]
```

ORACLE

Saving data

```
const table = schema.getTable('sessions_sql');
const collection = schema.getCollection('sessions_nosql');

// single insert
await table.insert('_id', 'title').values('DEV5985', 'This talk!').execute();

// batching inserts

const batchA = collection
  .add({ _id: 'DEV5986', title: 'Node.js and the MySQL Document Store', speakers: ['Rui Quelhas'] })
  .add({ _id: 'DEV6233', title: 'Connector/J Beyond JDBC...', speakers: ['Filipe Silva'] });

const batchB = table
  .insert(['_id', 'title'])
  .values(['DEV5959', 'Python and the MySQL Document Store']);

const batchC = table
  .insert('_id', 'title')
  .values('HOL1706', 'Developing Modern Applications with the MySQL Document Store and Node.js');

await Promise.all([batchA.execute(), batchB.execute(), batchC.execute()]);
```

ORACLE

Retrieving data

```
// const collection = schema.getCollection('sessions_nosql');

await collection.find('title LIKE :pattern')
  .bind({ pattern: '%Node.js%' })
  .fields('_id', 'speakers')
  .execute(console.log); // { _id: 'DEV5985', speakers: ['Rui Quelhas'] }


// const table = schema.getTable('sessions_sql');

await table.select() // project specific columns if needed
  .where('title LIKE :pattern')
  .bind('pattern', '%MySQL%')
  .orderBy('_id ASC')
  .limit(1)
  .offset(1)
  .execute(console.log); // ['HOL1706', 'Developing Modern Applications with...']
```

Updating data

```
// const collection = schema.getCollection('sessions_nosql');

await collection.modify('_id = :id')
  .bind('id', 'DEV5986')
  .set('title', 'The talk about Document Store!')
  .execute();

const doc = await collection.getOne('DEV5986');
console.log(doc); // { _id: 'DEV5986', title: 'The talk about Document Store!', speakers: ['Rui Quelhas] }


// const table = schema.getTable('sessions_sql');

await table.update()
  .where('_id = :id')
  .bind({ id: 'DEV5985' })
  .set('title', 'MySQL Connector/Node.js and the X DevAPI')
  .execute(console.log);

await table.select()
  .where('_id = :id')
  .bind('id', 'DEV5985')
  .execute(console.log) // ['DEV5985', 'MySQL Connector/Node.js and the X DevAPI']
```

Deleting data

```
// const collection = schema.getCollection('sessions_nosql');

await collection.remove('true').execute();

const docs = [];

const doc = await collection.find().execute(doc => docs.push(doc));
console.log(docs); // []


// const table = schema.getTable('sessions_sql');

await table.delete()
  .where('title LIKE :pattern')
  .bind('pattern', '%Node.js%')
  .execute();

await table.select()
  .execute(console.log) // ['DEV5959', 'Python and the MySQL Document Store']
```

Final thoughts

Some takeaways

- Simplify and secure your database installation
- Reap the benefits of both SQL and NoSQL
- High-availability and scalability via **InnoDB Cluster**
- Fast prototyping but always with maintainability in mind
- No language or platform lock-in (portable API constructs)
- Stick to a modern simple Node.js workflow (async, npm)
- Deploy everywhere where Node.js excells (containers, microservices, serverless and FaaS, desktop, IoT, etc.)
- It’s all open source

Links and Resources

- https://dev.mysql.com/doc/x-devapi-userguide/en/
- https://github.com/mysql/mysql-connector-nodejs
- https://www.npmjs.com/package/@mysql/xdevapi
- https://dev.mysql.com/doc/dev/connector-nodejs/8.0/
- https://www.mysql.com/news-and-events/web-seminars/mysql-document-store-and-node-js/
- https://www.mysql.com/news-and-events/web-seminars/mysql-document-store-and-node-js/

Q&A

Thank you!