JDBC

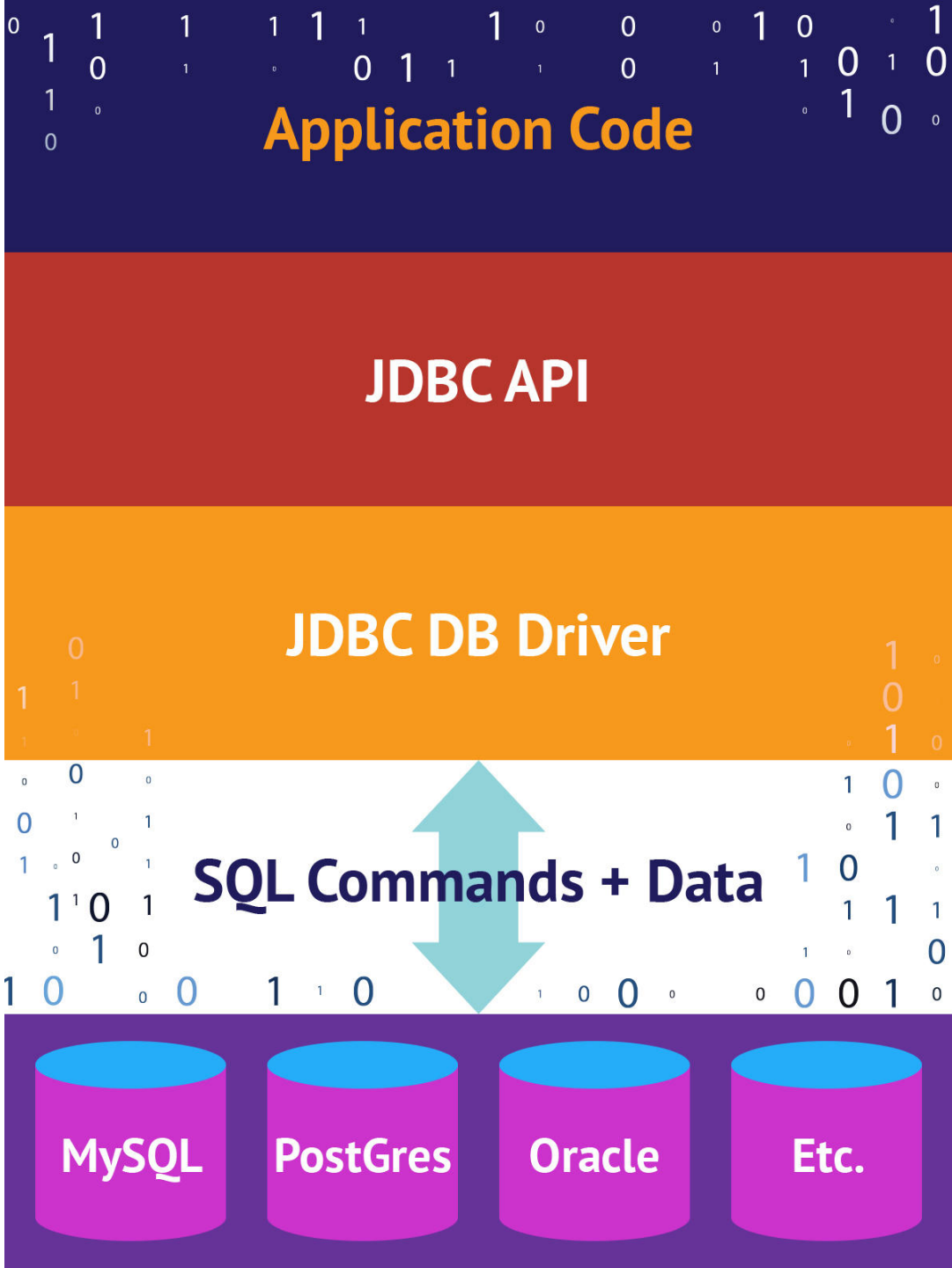
Java Database Connectivity

What is JDBC?

- Java API, which provides universal data access
  - API = Application Programming Interface
- With *JDBC* you can access almost any data sources
  - relational databases
  - spreadsheets
  - *JSON* files
- We will focus on accessing **relational databases**
- *JDBC* can work with any database (as long as proper drivers are provided)

What is a JDBC Driver?

- allows connecting to a certain database (e.g. *PostgreSQL*)
- converts *JDBC* calls into database calls
- implements a protocol for transferring data between client and database
  - statements (from client to database)
  - results (from database to client)



JDBC Driver Dependency

For accessing a *PostgreSQL* database, we need the following dependency in *pom.xml*:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.18</version>
</dependency>
```

What is a DataSource?

- An object that represents a particular DBMS (e.g. our *PostgreSQL* database)
  - DBMS = DataBase Managment System
- Each *JDBC* driver includes at least a basic `DataSource` implementation
  - Such a basic implementation doesn't provide connection pooling
- A connection pool is a cache of database connection objects
  - They are reused instead of created each time a connection is requested
- Many *JDBC* drivers also support connection pooling

Creating a DataSource

```
public DataSource createDataSource() {
    PGSimpleDataSource dataSource = new PGSimpleDataSource();
    dataSource.setServerNames(new String[]{"localhost"});
    dataSource.setDatabaseName("postgres");
    dataSource.setUser("postgres");
    dataSource.setPassword("postgres");
    dataSource.setSchema("plugin_manager_dev");
    return dataSource;
}
```

Please note: Never use hardcoded database properties in an application, you are going to deploy into production. Why do you think this would be a bad idea?

Establishing a Connection

- `DataSource.getConnection()` tries to establish a connection with the database
  - the method returns a `Connection` object
- The `Connection` should be used in a *try-with-resources* block
  - The `Connection` interface extends `AutoCloseable`
  - So we don't have to deal with closing the database connection

```
try (Connection connection = dataSource.getConnection()) {
    // use the connection in this block
    // ...
}
```

Executing SQL Statements

- With a `PreparedStatement` you can send SQL instructions to the database
- A `PreparedStatement` can be created with the `Connection` object

```
// sql is a String, which contains a SQL statement
PreparedStatement statement = connection.prepareStatement(sql);
```

- `PreparedStatement` objects contain precompiled SQL statements
  - At the first call the database compiles and caches the statement
  - If it is called again, the database can use the cached statement
  - That is even the case, if the parameter values change

Executing a DELETE

```
String sql = "delete from plugin where id=?";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setInt(1, plugin.getId());

if (statement.executeUpdate() == 0) {
    throw new SQLException("Delete failed: No rows affected");
}
```

Please Note:

- An *SQL* statement can contain one or more '?' parameter placeholders
- `executeUpdate()` returns the number of affected data rows

Executing an UPDATE

```
String sql = "update plugin set name=?, enabled=? where id=?";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setString(1, plugin.getName());
statement.setBoolean(2, plugin.isEnabled());
statement.setInt(3, plugin.getId());

if (statement.executeUpdate() == 0) {
    throw new SQLException("Update failed: No rows affected");
}
```

Executing an INSERT

```
String sql = "insert into plugin (name, enabled) values (?, ?)";
PreparedStatement statement =
    connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
statement.setString(1, name);
statement.setBoolean(2, enabled);

if (statement.executeUpdate() == 0) {
    throw new SQLException("Insert failed: No rows affected");
}
```

Please Note:

- The argument `RETURN_GENERATED_KEYS` is given to `prepareStatement()`
- Therefore the `PreparedStatement` can retrieve auto-generated keys

Retrieving Generated Keys

```
ResultSet keys = statement.getGeneratedKeys();
if (keys.next()) {
    return new Plugin(keys.getInt("id"), name, enabled);
} else {
    throw new SQLException("Insert failed: No key provided");
}
```

Please Note:

- The `next()` method is called once
- So the cursor of the `ResultSet` is moved to the first row
- Afterwards the key can be accessed via the column name

Executing a SELECT

```
String sql = "select id, name, enabled from plugin";
PreparedStatement statement = connection.prepareStatement(sql);
ResultSet results = statement.executeQuery();
```

Please Note:

- *INSERT*, *UPDATE* and *DELETE* statements are executed via `executeUpdate()`
- *Select* statements are executed via `executeQuery()`

Retrieving SELECT Results

```
while (results.next()) {
    int id = results.getInt("id");
    String name = results.getString("name");
    boolean enabled = results.getBoolean("enabled")
    // do something with those values
}
```

Please Note:

- The `next()` method returns false when there are no more rows in the `ResultSet`
- Therefore it can be used in a while loop to iterate through the `ResultSet`

Any questions?