COMP S380F Lecture 9: Spring JDBC Template

Dr. Keith Lee

School of Science and Technology

Hong Kong Metropolitan University

Overview of this lecture

- **Spring data sources**
  - JDBC driver-based, JNDI, Pooled, Embedded
- **Spring profiles** feature
- Using **JDBC** with **embedded database**:
  - Guest book example: *HelloSpringJDBC*

- JDBC problems:
  1. Boilerplate code
  2. Meaningless catch block for SQLException
- Spring data access template
- **Spring JDBC template**:
  - Guest book example: *HelloSpringJDBCTemplate*

Data source 1: JDBC driver-based data source

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="org.apache.derby.jdbc.ClientDriver"
    p:url="jdbc:derby://localhost:1527/account"
    p:username="nbuser"
    p:password="nbuser" />
```

xmlns:p="http://www.springframework.org/schema/p" (for property)

- The simplest data source you can configure in Spring is one that's defined through a JDBC driver:
  - **DriverManagerDataSource**:
    - Return a new connection every time a connection is requested.
    - Incur a performance cost for creating many new connections.
  - **SimpleDriverDataSource**: Similar to DriverManagerDataSource.
  - **SingleConnectionDataSource**:
    - Return the same connection every time a connection is requested.
    - Does not work well in multi-threaded applications.

Data source 2: JNDI data source

- Java EE application servers allow you to configure data sources to be retrieved via **JNDI (Java Naming and Directory Interface)**.
  - JNDI is a directory service that allows Java software to discover and look up data and objects via a name.
- **Benefits:**
  - JNDI data sources can be managed completely **external to the application**, allowing the application to ask for a data source when it's ready to access the database.
  - Data sources managed in an application server are often **pooled** for greater performance (i.e., it draws its connection from a database connection pool) and can be hot-swapped by system administrators.
- We can configure a reference to JNDI data source as if it were just another Spring bean using the <jee:jndi-lookup> from Spring's jee namespace.

```
<jee:jndi-lookup id="dataSource"  jndi-name="/jdbc/UserDS"   resource-ref="true" />
```

name of the resource in JNDI

For some application server/web container, e.g., Tomcat, the value given in jndi-name should be prepended with java:comp/env/

Data source 3: Pooled data source

- Spring supports Database Connection Pools to share a pool of open connections:
  - Apache Commons DBCP 2 (http://commons.apache.org/proper/commons-dbcp)
  - c3p0 (http://sourceforge.net/projects/c3p0)
  - HikariCP (https://github.com/brettwooldridge/HikariCP)
- E.g., DBCP's BasicDataSource:

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
   p:driverClassName="org.h2.Driver"
   p:url="jdbc:h2:tcp://localhost/~/user"
   p:username="sa"
   p:password=""
   p:initialSize="5"
   p:maxTotal="10" />
```

JDBC driver for the H2 database

The pool starts with 5 connections.

If more connections are needed,
at most 10 active connections can be created.

- BasicDataSource's pool-configuration properties:
  `initialSize, maxTotal, maxIdle, maxOpenPreparedStatements, maxWaitMillis, minEvictableIdleTimeMillis, minIdle, poolPreparedStatements`

Ref: https://commons.apache.org/proper/commons-dbcp/configuration.html

Data source 4: Embedded database

- An embedded database runs as part of your application instead of as a separate database server that your application connects to.
- Not suitable for production but perfect for development & testing:
  - You can populate your database with test data, and reset the database every time you restart your application or run your tests.
- E.g.,

```
<jdbc:embeddeddatabase id="dataSource" type="H2">
  <jdbc:script location="classpath:schema.sql"/>
  <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

- The **type** attribute indicates the type of the embedded database.
  - E.g., type="H2" or type="DERBY"
- You may configure zero or more <jdbc:script> elements to set up the database, e.g.,
  - schema.sql contains SQL to create the tables in the database
  - test-data.sql populates the database with test data.

Spring profiles

- We may need different data source beans in different environment.
  - E.g., development, quality assurance (QA), production.
- We can set different Spring profile for different data source.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans …>
  …
  <beans profile="dev">
    <jdbc:embeddeddatabase id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
    </jdbc:embeddeddatabase>
  </beans>
  <beans profile="qa">
    <bean id="dataSource“ class="org.apache.commons.dbcp.BasicDataSource"
        p:driverClassName="org.h2.Driver"
        p:url="jdbc:h2:tcp://localhost/~/user"
        p:username="sa“   p:password=""  p:initialSize="5"  p:maxTotal="10" />
  </beans>
  <beans profile="production">
    <jee:jndi-lookup id="dataSource" jndi-name="/jdbc/UserDS" resource-ref="true" />
  </beans>
</beans>
```

Spring profiles (cont')

web.xml

```
<web-app …>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>
<context-param>
  <param-name>spring.profiles.default</param-name>
  <param-value>dev</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>spring.profiles.default</param-name>
    <param-value>dev</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Set default profile for context

Set default profile for servlet

Spring profiles: Example

- Webapp example: ***HelloSpringJDBC***

security.xml

```
<b:beans …>
   <http auto-config="true"> … </http>
   <authentication-manager>…</authentication-manager>

   <b:beans profile="dev">
      <jdbc:embedded-database id="dataSource" type="DERBY">
         <jdbc:script location="classpath:create_user.sql" />
         <jdbc:script location="classpath:create_guestbook.sql" />
      </jdbc:embedded-database>
   </b:beans>


   <b:beans profile="qa">
      <b:bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
            p:driverClassName="org.apache.derby.jdbc.ClientDriver"
            p:url="jdbc:derby://localhost:1527/account"
            p:username="nbuser"
            p:password="nbuser" />
   </b:beans>
</b:beans>
```

For profile "**dev**", we use an embedded Derby database. We can use "**classpath:**" to access resource files.

For profile "**qa**", we use NetBean's Derby database.

Spring profiles: Example (cont')

- In the deployment descriptor (web.xml), we add a context parameter "spring.profiles.default" for selecting a profile.

web.xml

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/spring/*.xml</param-value>
</context-param>

<context-param>
   <param-name>spring.profiles.default</param-name>
   <param-value>qa</param-value>
</context-param>

<listener>
   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

- If you use the profile "**dev**", the database is reset when you deploy the webapp.
- If you use the profile "**qa**", the data in the database is persisted.

Typical JDBC Scenario

1. Load JDBC driver for specific RDBMS (`Driver` class)
2. Get connection to a named database (`Connection` class)
3. Set up a SQL query (or update) (`Statement` class)
4. Execute the query to obtain results (`ResultSet` class)
5. Iterate over the results in the `ResultSet`

JDBC interfaces (Recap from last lecture)

- **`java.sql.Statement`**
  - Represent a SQL statement (SELECT or UPDATE) to be sent to DBMS
  - Related methods:
    - Execution: `execute, executeQuery, executeUpdate`
    - Creation: `Connection.createStatement`
- **`java.sql.ResultSet`**
  - Hold the result of executing an SQL query (i.e., the result relation)
  - Handle access both to rows and columns within rows
  - Related methods:
    - Iteration: `next`
    - Accessing data: `get{Type}(position|name)`
      - E.g., `getInt(4), getString("name")`

PreparedStatement object (Recap from last lecture)

- A more realistic case is that the same kind of SQL statement is processed over and over (rather than a static SQL statement).

```
SELECT * FROM employee WHERE id = 3;
SELECT * FROM employee WHERE id = 7;
SELECT * FROM employee WHERE id = 25;
SELECT * FROM employee WHERE id = 21;
...
SELECT * FROM employee WHERE id = ?
```

- In PreparedStatement, a placeholder (?) will be bound to an incoming value before execution (no recompilation).

```
PreparedStatement ps =
       conn.prepareStatement("SELECT * FROM employee WHERE id=?");
ResultSet rs;
for (int i = 0; i < 1000; i++) {
   ps.setInt(1, i);
   rs = ps.executeQuery();
   /* Do something more */
}
```

Transaction management

- By default, JDBC commits each update when you call `executeUpdate()`.
- Committing after each update can be suboptimal in terms of performance.
- It is also not suitable if you want to manage a series of operations as a logical single operation (i.e., transaction).

```
conn.setAutoCommit(false); // this marks START_TRANSACTION
Statement stmt = conn.createStatement();
try {
   stmt.executeUpdate("UPDATE ACCOUNTS SET bal=bal-100 WHERE id=101");
   stmt.executeUpdate("UPDATE ACCOUNTS SET bal=bal+100 WHERE id=102");
   conn.commit();
} catch (Exception e) {
   conn.rollback();
}
```

GuestBook example with JDBC

- Webapp example: ***HelloSpringJDBC***
- The following database table stores all guestbook entries.

Other Sources > create_guestbook.sql

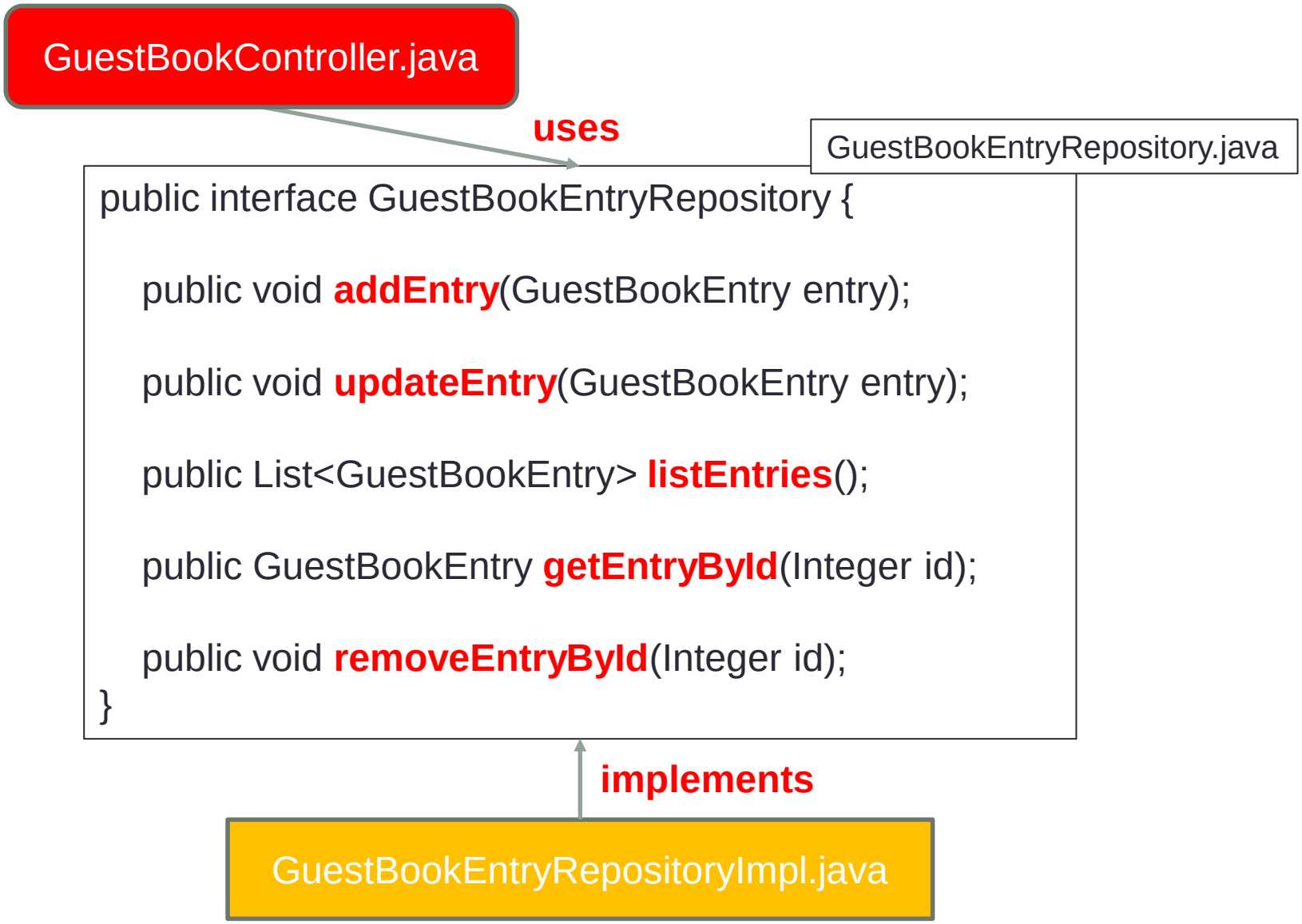
```
CREATE TABLE guestbook (
    id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1,
                                                      INCREMENT BY 1),
    name VARCHAR(50),
    message VARCHAR(200),
    date TIMESTAMP,
    PRIMARY KEY (id)
);
```

TIMESTAMP keeps both date and time information.

dispatcher-servlet.xml

```
<context:component-scan base-package="hkmu.comps380f.controller,
                                      hkmu.comps380f.dao" />
```

GuestBook example: Repository interface



GuestBook example: GuestBookController

- **@Resource** and **@Autowired** automatically find the matched Spring bean (by type) and set it as gbEntryRepo.

GuestBookController.java

```
@Controller
@RequestMapping("/guestbook")
public class GuestBookController  {
    @Resource
    private GuestBookEntryRepository gbEntryRepo;

    @GetMapping({"", "/view"})
    public String index(ModelMap model) {
        model.addAttribute("entries", gbEntryRepo.listEntries());
        return "GuestBook";
    }

    // …
    @PostMapping("/add")
    public View addCommentHandle(GuestBookEntry entry) {
        entry.setDate(new Date());
        gbEntryRepo.addEntry(entry);
        return new RedirectView("/guestbook/view", true);
    }
    // …
```

GuestBook example: GuestBookController (cont')

GuestBookController.java

```
    // …
    @GetMapping("/edit")
    public String editCommentForm(@RequestParam("id") Integer entryId,
                                  ModelMap model) {
        model.addAttribute("entry", gbEntryRepo.getEntryById(entryId));
        return "EditComment";
    }

    @PostMapping("/edit")
    public View editCommentHandle(GuestBookEntry entry) {
        entry.setDate(new Date());
        gbEntryRepo.updateEntry(entry);
        return new RedirectView("/guestbook/view", true);
    }

    @GetMapping("/delete")
    public String deleteEntry(@RequestParam("id") Integer entryId) {
        gbEntryRepo.removeEntryById(entryId);
        return "redirect:/";
    }

}
```

GuestBook example: Repository implementation

- We use JDBC to implement the repository methods.
- **@Repository** tells Spring to create a Spring bean for this class.

GuestBookEntryRepositoryImpl.java

```
@Repository
public class GuestBookEntryRepositoryImpl implements GuestBookEntryRepository {

    @Autowired
    DataSource dataSource;

    //  implementation for repository methods
}
```

- We use **@Autowired** again so that Spring will automatically set dataSource to be the Spring data source bean.

Repository implementation: addEntry

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_INSERT_ENTRY
        = "insert into guestbook (name, message, date) values (?, ?, ?)";
@Override
public void addEntry(GuestBookEntry entry) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_INSERT_ENTRY);
        stmt.setString(1, entry.getName());
        stmt.setString(2, entry.getMessage());
        stmt.setTimestamp(3, new Timestamp(entry.getDate().getTime()));
        stmt.executeUpdate();
    } catch (SQLException e) {
        // do something ... not sure what, though
    } finally {
        try {
            if (stmt != null) { stmt.close(); }
            if (conn != null) { conn.close(); }
        } catch (SQLException e) {
            // Even less sure about what to do here
        }
    }
}
```

Convert the `java.util.Date` object to a `java.sql.Timestamp` object.

Problems of using JDBC

Problem 1:

- In the previous `addEntry()` function, we need a lot of code to insert an object into a database, and it is about as simple as it gets.
- Only a few lines of code actually do the insert.
- Other lines of codes are **boilerplate code** for controlling transactions, managing resources, and handling exceptions.

Problem 2:

- Some common problems that cause `SQLException` to be thrown:
  - Unable to connect to database, SQL syntax error, tables/columns are non-existent, values to be inserted/updated violates DB constraints
- Most of these are fatal condition and cannot be remedied in a catch block.
- The `SQLException` catching code is actually useless.

Repository implementation: listEntries

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_SELECT_ALL_ENTRY
        = "select id, name, message, date from guestbook";
@Override
public List<GuestBookEntry> listEntries() {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_SELECT_ALL_ENTRY);
        rs = stmt.executeQuery();
        List<GuestBookEntry> entries = new ArrayList<>();
        while (rs.next()) {
            GuestBookEntry entry = new GuestBookEntry();
            entry.setId(rs.getInt("id"));
            entry.setName(rs.getString("name"));
            entry.setMessage(rs.getString("message"));
            entry.setDate(toDate(rs.getTimestamp("date")));
            entries.add(entry);
        }
        return entries;
    } catch (SQLException e) {
    // Lots of boilerplate code omitted …
}
```

```
public static Date toDate(Timestamp timestamp) {
  long milliseconds = timestamp.getTime()
          + (timestamp.getNanos() / 1000000);
  return new Date(milliseconds);
}
```

Repository implementation: getEntryById

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_SELECT_ENTRY
        = "select id, name, message, date from guestbook where id = ?";
@Override
public GuestBookEntry getEntryById(Integer id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_SELECT_ENTRY);
        stmt.setInt(1, id);
        rs = stmt.executeQuery();
        GuestBookEntry entry = null;
        if (rs.next()) {
            entry = new GuestBookEntry();
            entry.setId(rs.getInt("id"));
            entry.setName(rs.getString("name"));
            entry.setMessage(rs.getString("message"));
            entry.setDate(toDate(rs.getTimestamp("date")));
        }
        return entry;
    } catch (SQLException e) {
    // Lots of boilerplate code omitted …
}
```

Repository implementation: updateEntry

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_UPDATE_ENTRY
        = "update guestbook set name = ?, message = ?, date = ? where id = ?";
@Override
public void updateEntry(GuestBookEntry entry) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_UPDATE_ENTRY);
        stmt.setString(1, entry.getName());
        stmt.setString(2, entry.getMessage());
        stmt.setTimestamp(3, new Timestamp(entry.getDate().getTime()));
        stmt.setInt(4, entry.getId());
        stmt.executeUpdate();
    } catch (SQLException e) {
    } finally {
        try {
            if (stmt != null) { stmt.close(); }
            if (conn != null) { conn.close(); }
        } catch (SQLException e) {
        }
    }
}
```

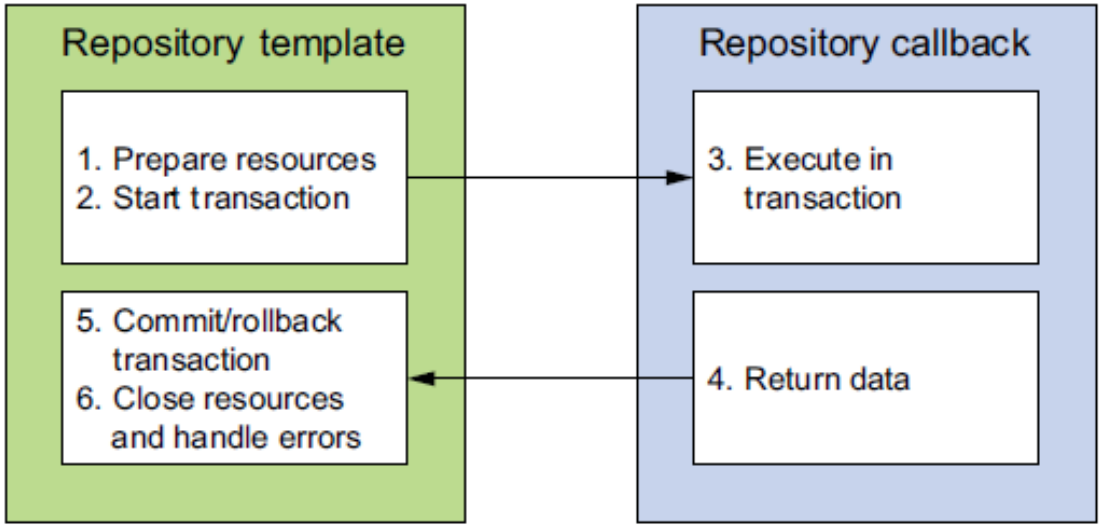
Repository implementation: removeEntryById

GuestBookEntryRepositoryImpl.java

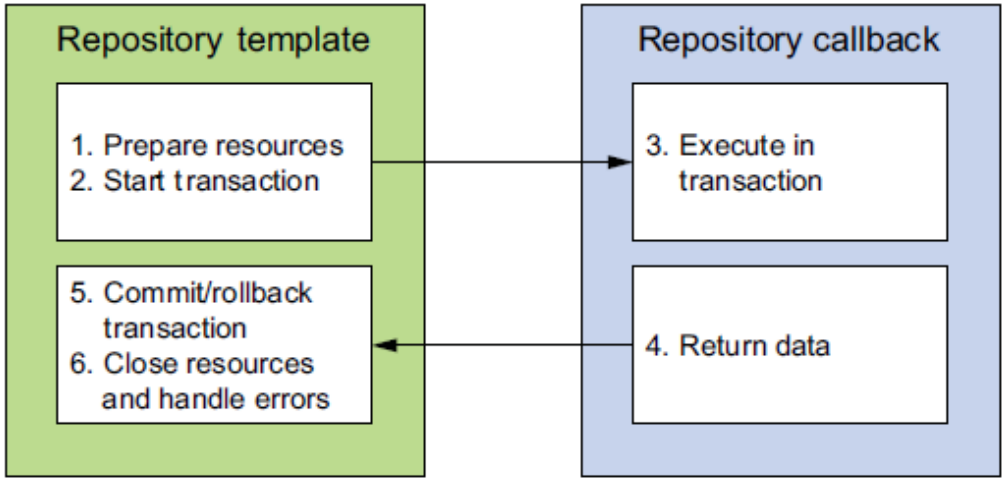
```
private static final String SQL_DELETE_ENTRY
        = "delete from guestbook where id = ?";
@Override
public void removeEntryById(Integer id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(SQL_DELETE_ENTRY);
        stmt.setInt(1, id);
        stmt.executeUpdate();
    } catch (SQLException e) {
    } finally {
        try {
            if (stmt != null) { stmt.close(); }
            if (conn != null) { conn.close(); }
        } catch (SQLException e) {
        }
    }
}
```

Spring data access template

- No matter what persistence technology you are using, certain data access steps are required.
- Spring separates the fixed and variable parts of the data access process into two distinct classes:
  - **Templates:** Manage the fixed part of the process
  - **Callbacks:** Handle your custom data access code



Spring data access template (cont')



- **Template classes** handle the fixed parts of data access, e.g., controlling transactions, managing resources, and handling exceptions.
- **Callback** is implemented by developers, which include creating statements, binding parameters and marshalling result sets.
- Spring comes with several data access templates, e.g., `CciTemplate`, `JdbcTemplate`, `NamedParameterJdbcTemplate`, `SimpleJdbcTemplate`, `HibernateTemplate`, `JpaTemplate`, `SqlMapClientTemplate`, `JdoTemplate`.

Spring JDBC template: Example

- We will use `JdbcTemplate`, which performs JDBC operations more easily by removing repetitive data access code blocks (the boilerplate code), properly handling resource clean-ups, etc.
- Webapp example: ***HelloSpringJDBCTemplate***
- Compared to HelloSpringJDBC, the only difference is the repository implementation class:

GuestBookEntryRepositoryImpl.java

```
@Repository
public class GuestBookEntryRepositoryImpl implements GuestBookEntryRepository {

    private final JdbcOperations jdbcOp;

    @Autowired
    public GuestBookEntryRepositoryImpl(DataSource dataSource) {
        jdbcOp = new JdbcTemplate(dataSource);
    }
    //  implementation for repository methods
}
```

Using @Autowired on the constructor automatically binds the arguments.

Spring JDBC template: RowMapper

- **RowMapper** is a useful interface provided by Spring JDBC.
- It can map rows in a ResultSet to an object on per row basis.

GuestBookEntryRepositoryImpl.java

```
@Repository
public class GuestBookEntryRepositoryImpl implements GuestBookEntryRepository {

    //  implementation for repository methods

    private static final class EntryRowMapper implements RowMapper<GuestBookEntry> {

        @Override
        public GuestBookEntry mapRow(ResultSet rs, int i) throws SQLException {
            GuestBookEntry entry = new GuestBookEntry();
            entry.setId(rs.getInt("id"));
            entry.setName(rs.getString("name"));
            entry.setMessage(rs.getString("message"));
            entry.setDate(toDate(rs.getTimestamp("date")));
            return entry;
        }
    }
}
```

Implementation: addEntry

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_INSERT_ENTRY
        = "insert into guestbook (name, message, date) values (?, ?, ?)";

@Override
public void addEntry(GuestBookEntry entry) {

    jdbcOp.update(SQL_INSERT_ENTRY,
            entry.getName(),
            entry.getMessage(),
            new Timestamp(entry.getDate().getTime())
    );

}
```

Implementation: listEntries, getEntryById

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_SELECT_ALL_ENTRY
        = "select id, name, message, date from guestbook";
@Override
public List<GuestBookEntry> listEntries() {
    return jdbcOp.query(SQL_SELECT_ALL_ENTRY, new EntryRowMapper());
}
```

```
private static final String SQL_SELECT_ENTRY
        = "select id, name, message, date from guestbook where id = ?";
@Override
public GuestBookEntry getEntryById(Integer id) {
    return jdbcOp.queryForObject(SQL_SELECT_ENTRY, new EntryRowMapper(), id);
}
```

Implementation: updateEntry, removeEntryById

GuestBookEntryRepositoryImpl.java

```
private static final String SQL_UPDATE_ENTRY
        = "update guestbook set name = ?, message = ?, date = ? where id = ?";
@Override
public void updateEntry(GuestBookEntry entry) {
    jdbcOp.update(SQL_UPDATE_ENTRY,
            entry.getName(),
            entry.getMessage(),
            new Timestamp(entry.getDate().getTime()),
            entry.getId());
}
```

```
private static final String SQL_DELETE_ENTRY
        = "delete from guestbook where id = ?";
@Override
public void removeEntryById(Integer id) {
    jdbcOp.update(SQL_DELETE_ENTRY, id);
}
```