rule FINANCIAL

The Sector Specialists

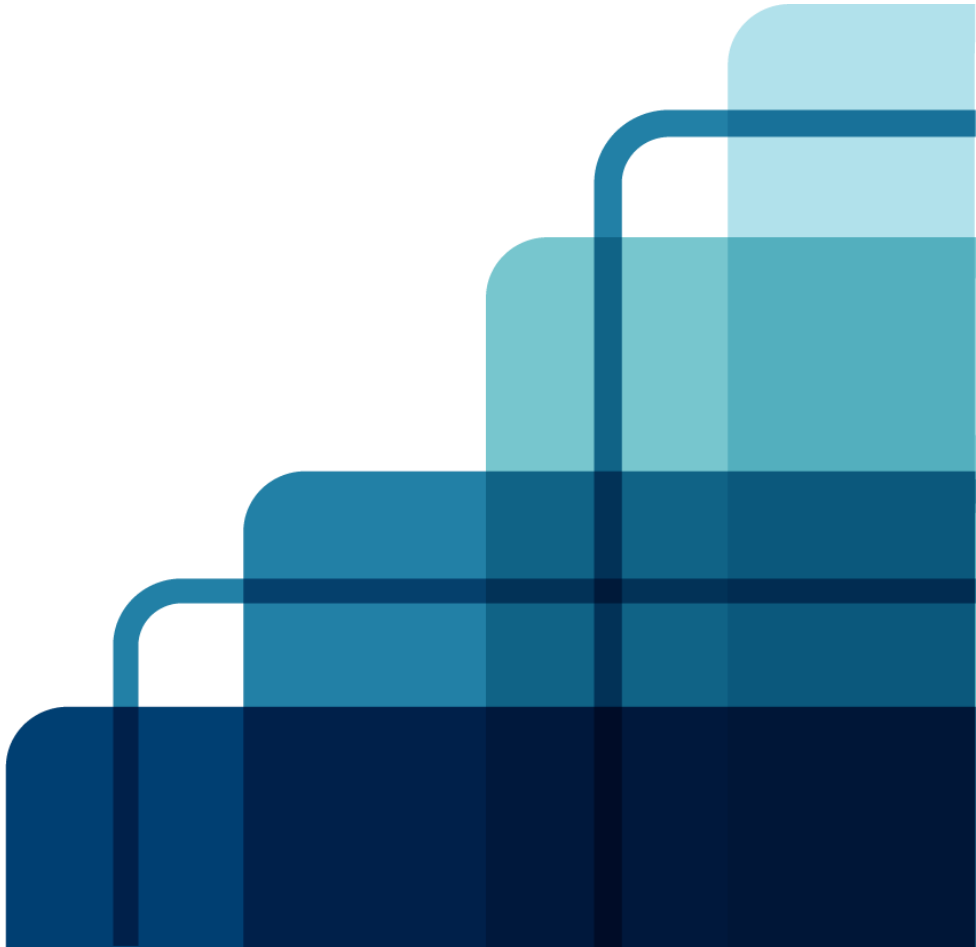
MyBatis

Overview

Prepared by: Piotr Kosmowski

Submitted on: 17.01.2013

Version: 1.0



Who am I?

Piotr Kosmowski

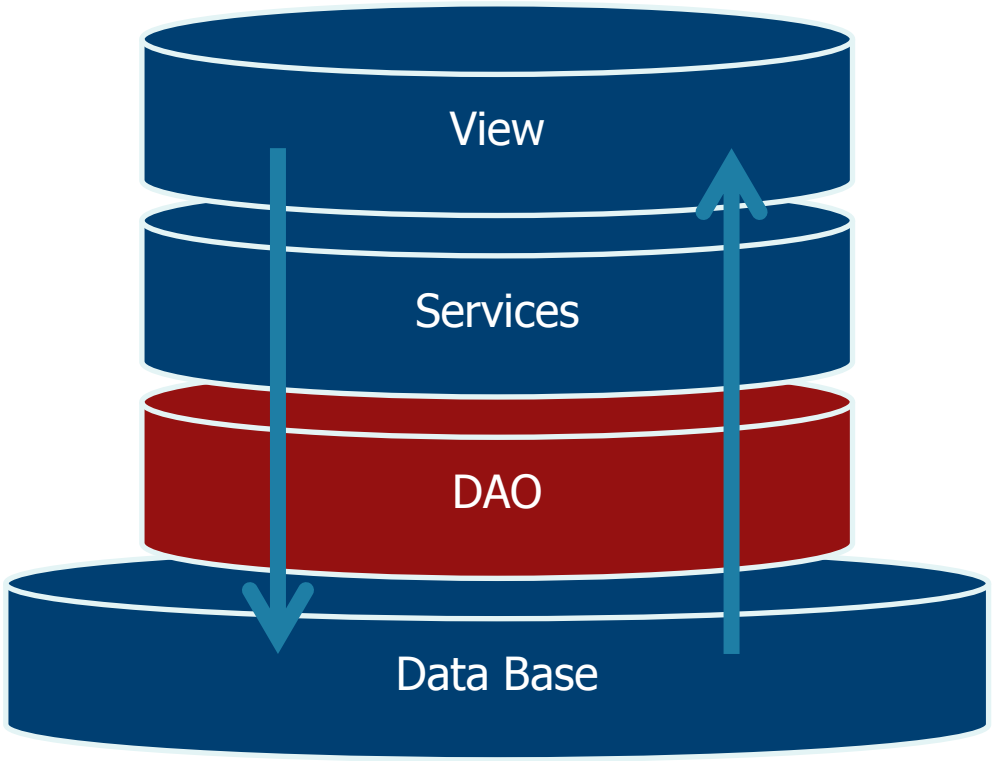
- Java Developer
- Experience: 6yrs
- Works for: Rule Financial

Road map

- Existing technologies of implementing DAO layer
- About MyBatis project
- How does it works – overview
- Simple project structure on example
- Configuration
- Xml way
  - Queries
  - Mappers
  - Dynamic language
- Anotation way
- Provider way
- Sumarization
- Questions

DAO

- DAO (Data Access Object) – an object / layer that provides an abstract interface to some storage – Relational Data Base.



DAO implementations
Technologies, frameworks, libraries

- SQL (Structured Query Language) – a language that is used to comunicate with Relational Data Bases
- Java -> JDBC -> Driver

- JDBC
  - Pure JDBC
  - SpringDAO (http://www.springsource.org/)
  - DbUtils (http://commons.apache.org/dbutils/)
  - Own monster
  - **MyBatis**

- ORM (Object-relational mapping)
  - JPA (1.0, 2.0)
    - Hibernate [1.0] (http://www.hibernate.org/)
    - EclipseLink [2.0] (http://www.eclipse.org/eclipselink/)
    - TopLink (http://www.oracle.com/technetwork/middleware/toplink/overview/index.html)
    - OpenJPA (http://openjpa.apache.org/)

- JDO
- NoSQL

JDBS

"Back to the Future"

```
Connection conn = null;
Statement stmt = null;
try {
    Class.forName("com.mysql.jdbc.Driver");
    conn = DriverManager.getConnection("jdbc:mysql://localhost/EMP", "username", "password");
    stmt = conn.createStatement();
    String sql = "SELECT id, first, last, age FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);
    while (rs.next()) {
        int id = rs.getInt("id");
        int age = rs.getInt("age");
        String first = rs.getString("first");
        String last = rs.getString("last");
    }
    rs.close();
    stmt.close();
    conn.close();
} catch (Exception e) {
    e.printStackTrace();
```

```
} finally {
        try {
            if (stmt != null) {
                stmt.close();
            }
        } catch (SQLException se2) {
        }
        try {
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException se) {
            se.printStackTrace();
        }
    }
```

MyBatis
Overview

MyBatis

- What is MyBatis?
  - It is data mapper framework that makes easier to use a relational database with OO applications.
  - MyBatis couples objects with SQL or stored procedures statements using a XML descriptor or annotations.
  - Simplicity is the biggest advantage of the MyBatis data mapper over object relational mapping tools.

- About project
  - Homepage: http://www.mybatis.org
  - Licence: The Apache Software License, Version 2.0
  - Foundation: ASM -> Google
  - Versions for: Java, .NET, Scala, Ruby

MyBatis
How does it works?

MyBatis

DAO - interface

```
public List<Post> findPostsByKeyword(String key)
```

QUERY

```
SELECT * FROM post p WHERE p.name = #{keyword}
```

DB

MAPPER

MODEL

```
public class Post{
  Long id;
  String tittle;
  String content;
  Date date;
  ...
}
```

RESULT SET

ID	TITLE	CONTENT	DATE
11	Hello	World	01-01-2013
25	Good	Afternoon	02-01-2013
37	Bye	Everyone	03-01-2013

Essential elements
For cooking

MyBatis

- Model (java)
- DAO interface (java)
- Queries / Mappers (xml / anotations / java)
- Configuration (xml / java)
  - DataSources – sources of data
    - Unpooled (single connection)
    - Pooled
    - JNDI
  - TransactionManager
  - Queries
  - Mappers
  - Aliases

rule FINANCIAL

Simple project

Simple project
Configuration – XML way

MyBatis

- Configuration (mybatis-config.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <properties>
        <property name="driver" value="org.apache.derby.jdbc.ClientDataSource"/>
        <property name="url" value="jdbc:derby://localhost:1527/sun-appserv-samples"/>
        <property name="username" value="APP"/>
        <property name="password" value="APP"/>
    </properties>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="org/mybatis/example/BlogDao.xml"/>
    </mappers>
</configuration>
```

Simple project
Configuration – Java way

MyBatis

- Configuration (MyBatisConfigurationFactory.java)

```
DataSource dataSource = DataSourceFactory.getDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment = new Environment("development",transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogDao.class);
```

- DataSource factory (DataSourceFactory.java)

```
ComboPooledDataSource ds = new ComboPooledDataSource();
       ds.setDriverClass("org.apache.derby.jdbc.ClientDataSource");
       ds.setJdbcUrl("jdbc:derby://localhost:1527/sun-appserv-samples");
       ds.setUser("APP");
       ds.setPassword("APP");
```

Simple project
Mapper / DAO

MyBatis

- Query

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd">
<mapper namespace="org.mybatis.example.BlogDao">
    <select id="findAllBlogs" resultType="org.mybatis.example.Blog">
        SELECT b.id, b.name FROM app.blog b
    </select>
    <insert id="insertBlog" parameterType="org.mybatis.example.Blog">
        INSERT INTO app.blog(id,name) values(#{id},#{name})
    </insert>
</mapper>
```

- DAO

```
public interface BlogDao {

    public List<Blog> findAllBlogs();

    public void insertBlog(Blog blog);

}
```

- Model

```
public class Blog {

    Long id;
    String name;
    List<Post> posts;

    ...
}
```

- No mapper required for mapping 1:1

Simple project
SqlSessionFactory

MyBatis

Every MyBatis application centers around an instance of **SqlSessionFactory**. A SqlSessionFactory instance can be acquired by using the **SqlSessionFactoryBuilder**. SqlSessionFactoryBuilder can build a SqlSessionFactory instance from an XML **configuration** file, of from a custom prepared instance of the Configuration class.

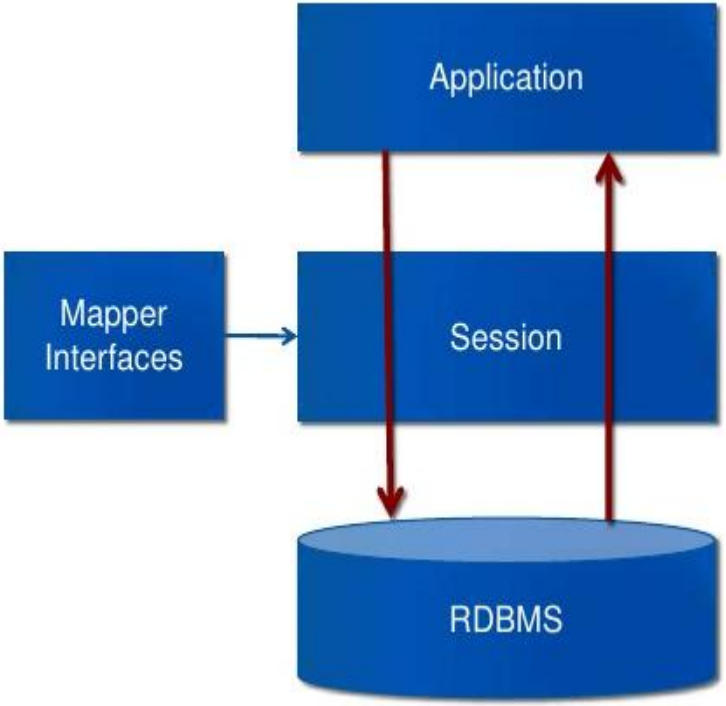
```
// init configuration and create SessionFactory
InputStream inputStream =
Resources.getResourceAsStream("mybatis-config.xml");
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

// open new session
SqlSession session = sqlSessionFactory.openSession();

// obtaining mapper - implementation of DAO's interface
List<Blog> personList = null;
BlogDao mapper = session.getMapper(BlogDao.class);

// execute query
try {
  personList = mapper.findAllBlogs();
} catch (Exception e) {
  session.close();
}

System.out.println(Joiner.on(",\n").join(personList));
```



Simple project
Example

QUERIES
Structure

MyBatis

- An SQL query that will be executed on data base
- Unlike in ORM queries mus be writen explicity
- Can be definied in XML file, adnotation or generated with builder
- Supported types:
  - insert
  - update
  - delete
  - select

- XML elements:
  - Type (insert, update, delete, select)
  - Id – name of the method

MyBatis

XML way

QUERIES in XML
Select

- To create queries that returns data unlike insert/update/delete
- Basic attributes:
  - **id** – reference to method name
  - **parameterType**
  - **resultType / resultMap** – expected type or mapper
- Other attributes: flushCache, useCache, timeout, fetchSize, statementType, resultSetType

- Sample:

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

Is equivalent to old way:

```
String selectPerson = "SELECT * FROM PERSON WHERE ID=?";
PreparedStatement ps = conn.prepareStatement(selectPerson);
ps.setInt(1,id);
```

QUERIES in XML
Parameters

MyBatis

- Parameters
  - If simple parameterType than any named parameter can be used
  - When complex type than mapping to its property

```
<select id="selectPerson" parameterType="User" resultType="hashmap">
    SELECT * FROM PERSON WHERE ID = #{id} and NAME like '%#{name}%' and SURNAME like '%#{surname}%'
</select>
```

  - Can specify more specific data type (jdbcType is required by JDBC if column is nullable)

```
#{property,javaType=int,jdbcType=NUMERIC}
```

  - Can use own type handler

```
#{age,javaType=int,jdbcType=NUMERIC,typeHandler=MyTypeHandler}
```

  - Strings are by default enclosed with quotes. To bypass this use **$** instead of **#** before parameter name

```
... ORDER BY ${columnName}
```

  - Advanced: IN, OUT, INOUT mode

MAPPERS in XML
ResultMap

MyBatis

- The resultMap element is the **most important and powerful** element in MyBatis. It's what allows you to do **away with 90% of the code** that JDBC requires to retrieve data from ResultSets, and in some cases allows you to do things that JDBC does not even support.
- In fact, to write the equivalent code for something like a join mapping for a complex statement could probably span thousands of lines of code.
- The design of the ResultMaps is such that **simple statements don't require explicit result mappings** at all, and more complex statements require no more than is absolutely necessary to describe the relationships.

```
<select id="selectUsers" parameterType="int" resultType="hashmap">
  select id, username, hashedPassword from some_table where id = #{id}
</select>
```

- This query will map result as **HashMap**, where columns are mapped as keys.
- This way is not convenient when using domain model
- MyBatis supports also JavaBeans / POJOS mappings

MAPPERS in XML
Direct model mapping

MyBatis

- Model

```
package com.someapp.model;

public class User {
  private int id;
  private String username;
  private String hashedPassword;

  ...
}
```

- Query with direct mapping to model class (use resultType)

```
<!--In Config XML file -->
<typeAlias type="com.someapp.model.User" alias="User"/>

<!--In SQL Mapping XML file -->
<select id="selectUsers" parameterType="int" resultType="User">
  select id, username, hashedPassword from some_table where id = #{id}
</select>
```

Based on the JavaBeans specification, the above class has 3 properties: id, username, and hashedPassword. These match up exactly with the column names in the select statement. Such a JavaBean could be mapped to a ResultSet just as easily as the HashMap.

By default MyBatis use public modifiers – setters and getters. Also possible are costructor and objectFactoryHandler.

MAPPERS in XML
Direct mapping

If the column names did not match exactly, you could employ select clause aliases (a standard SQL feature) on the column names to make the labels match. For example:

- Model

```
package com.someapp.model;

public class User {
  private int id;
  private String username;
  private String hashedPassword;

  ...
}
```

- Query with direct mapping to model class (use resultType)

```
<select id="selectUsers" parameterType="int" resultType="User">
  select
    user_id         as "id",
    user_name       as "userName",
    hashed_password as "hashedPassword"

  from some_table
    where id = #{id}
</select>
```

MAPPERS in XML
ResultMap

MyBatis

A way how to work with not direct mapping - **ResultMap**

- Model

```
package com.someapp.model;

public class User {
  private int id;
  private String username;
  private String hashedPassword;

  ...
}
```

- Query with direct mapping to model class (**use resultMap**)

```
<select id="selectUsers" parameterType="int" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table where id = #{id}
</select>
```

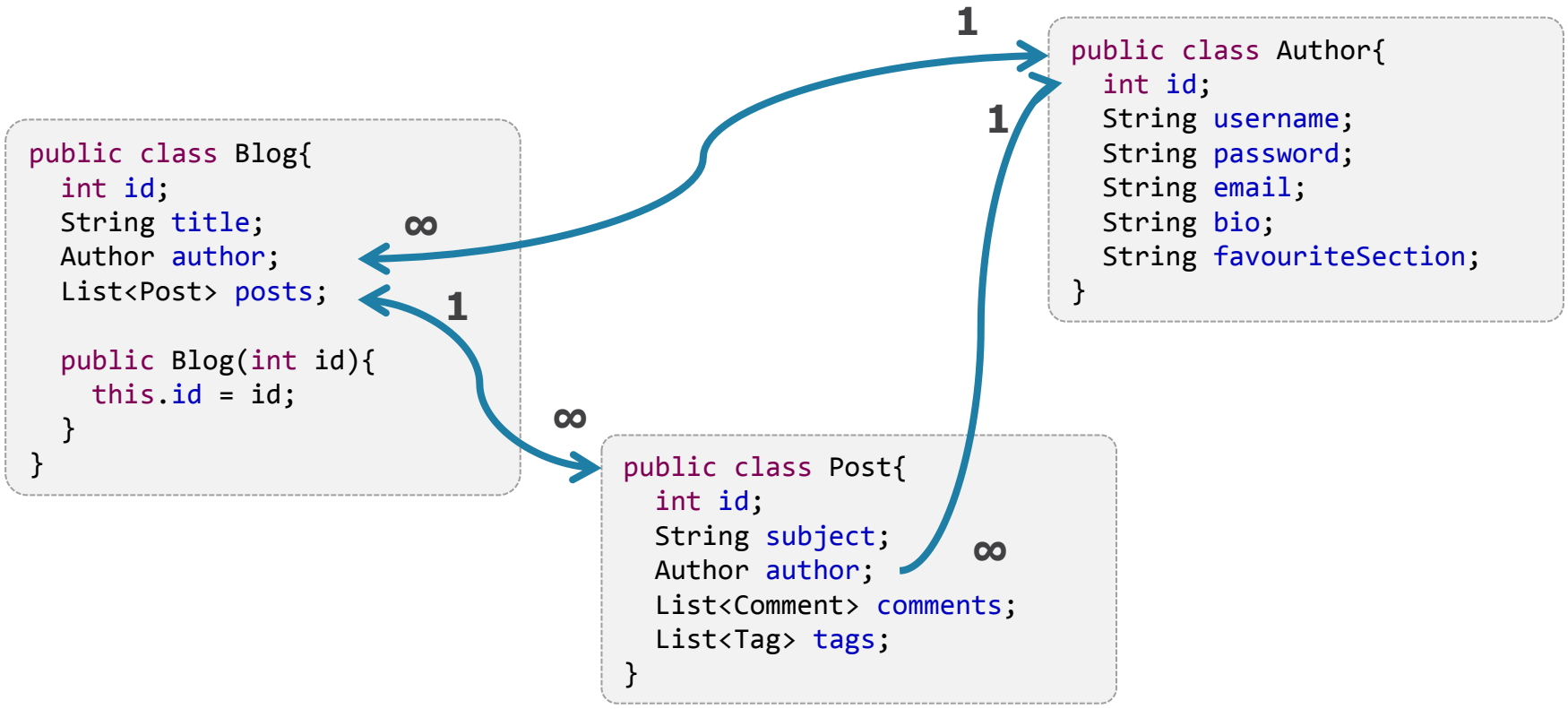
- Mapper

```
<resultMap id="userResultMap" type="User">
  <id property="id" column="user_id" />
  <result property="username" column="user_name" />
  <result property="hashedPassword" column="hashed_password" />
</resultMap>
```

MyBatis was created with one idea in mind: Databases aren't always what you want or need them to be and it would be great if it was possible to have a single database map perfectly to all of the applications that use it, it's not. Result Maps are the answer that MyBatis provides to this problem.

MAPPERS in XML

ResultMap - Advanced result mapping



MAPPERS in XML

ResultMap - Advanced result mapping

MyBatis

```
<!--Very Complex Statement -->
<select id="selectBlogDetails" parameterType="int"
resultMap="detailedBlogResultMap">
  select
    B.id as blog_id,
    B.title as blog_title,
    B.author_id as blog_author_id,
    A.id as author_id,
    A.username as author_username,
    A.password as author_password,
    A.email as author_email,
    A.bio as author_bio,
    A.favourite_section as author_favourite_section,
    P.id as post_id,
    P.blog_id as post_blog_id,
    P.author_id as post_author_id,
    P.created_on as post_created_on,
    P.section as post_section,
    P.subject as post_subject,
    P.draft as draft,
    P.body as post_body,
    C.id as comment_id,
    C.post_id as comment_post_id,
    C.name as comment_name,
    C.comment as comment_text,
    T.id as tag_id,
    T.name as tag_name
  from Blog B
    left outer join Author A on A.id = B.author_id
    left outer join Post P on P.blog_id = B.id
    left outer join Comment C on C.post_id = P.id
    left outer join Post_Tag PT on PT.post_id = P.id
    left outer join Tag T on PT.tag_id = T.id
  where B.id = #{id}
</select>
```

MAPPERS in XML

ResultMap - Advanced result mapping

MyBatis

```
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int" />
  </constructor>
  <result property="title" column="blog_title" />
  <association property="author" column="blog_author_id„ javaType=" Author">
    <id property="id" column="author_id" />
    <result property="username" column="author_username" />
    <result property="password" column="author_password" />
    <result property="email" column="author_email" />
    <result property="bio" column="author_bio" />
    <result property="favouriteSection" column="author_favourite_section" />
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id" />
    <result property="subject" column="post_subject" />
    <association property="author" column="post_author_id„ javaType="Author" />
    <collection property="comments" column="post_id" ofType=" Comment">
      <id property="id" column="comment_id" />
    </collection>
    <collection property="tags" column="post_id" ofType=" Tag">
      <id property="id" column="tag_id" />
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost" />
    </discriminator>
  </collection> </resultMap>
```

```
public class Blog{
  int id;
  String title;
  Author author;
  List<Post> posts;

  public Blog(int id){
    this.id = id;
  }
}
```

```
public class Author{
  int id;
  String username;
  String password;
  String email;
  String bio;
  String favouriteSection;
}
```

```
public class Post{
  int id;
  String subject;
  Author author;
  List<Comment> comments;
  List<Tag> tags;
}
```

MAPPERS in XML

ResultMap - Advanced result mapping

MyBatis

- Best Practices:
  - Always build ResultMaps incrementally
  - Unit tests really help out here
- If you try to build a gigantic resultMap like the one above all at once, it's likely you'll get it wrong and it will be hard to work with
- Start simple, and evolve it a step at a time.

rule FINANCIAL

MAPPERS in XML
Simple properties


```
<id property="id" column="author_id" />
<result property="username" column="author_username" />
<result property="password" column="author_password" />
<result property="email" column="author_email" />
<result property="bio" column="author_bio" />
<result property="favouriteSection" column="author_favourite_section" />
```

```
public class Author{
  int id;
  String username;
  String password;
  String email;
  String bio;
  String favouriteSection;
}
```

- These are the most basic of result mappings. Both id, and result map a single column value to a single property or field of a simple data type (String, int, double, Date, etc.).
- Id will flag the result as an identifier property to be used when comparing object instances. This helps to improve general performance, but especially performance of caching and nested result mapping (i.e. join mapping).

- Properties: propery, column, javatype, jdbcType, typeHandler

MAPPERS in XML
Constructor

- Constructor injection allows you to set values on a class upon instantiation, without exposing public methods.
- Often tables that contain reference or lookup data that rarely or never changes is suited to immutable classes.
- MyBatis also supports private properties and private JavaBeans properties to achieve this, but some people prefer Constructor injection.

```
<constructor>
   <idArg column="blog_id" javaType="int" />
</constructor>
```

```
public class Blog{
  int id;
  String title;
  Author author;
  List<Post> posts;

  public Blog(int id){
    this.id = id;
  }
}
```

MAPPERS in XML
Association


```
<association property="author" column="blog_author_id„ javaType=" Author">
    <id property="id" column="author_id" />
    <result property="username" column="author_username" />
    <result property="password" column="author_password" />
    <result property="email" column="author_email" />
    <result property="bio" column="author_bio" />
    <result property="favouriteSection" column="author_favourite_section" />
  </association>
```

```
public class Blog{
  int id;
  String title;
  Author author;
  List<Post> posts;

  public Blog(int id){
    this.id = id;
  }
}
```

- The association element deals with a "has`one" type relationship.
- Can handle association in two ways:
  - **Nested Results** - By using nested result mappings to deal with repeating subsets of joined results
  - **Nested Select** - By executing another mapped SQL statement that returns the complex type desired (**N+1 selects**)

```
<select id="selectBlog" parameterType="int" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>
<resultMap id="blogResult" type="Blog">
  <association property="author" column="blog_author_id" javaType="Author" select="selectAuthor"/>
</resultMap>
<select id="selectAuthor" parameterType="int" resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
</select>
```

MAPPERS in XML
Collections

MyBatis

```
 <collection property="posts" ofType="Post">
    <id property="id" column="post_id" />
    <result property="subject" column="post_subject" />
</collection>
```

```
public class Blog{
  int id;
  String title;
  Author author;
  List<Post> posts;

  public Blog(int id){
    this.id = id;
  }
}
```

- Works identically as associations
- Can be also mapped to nested select
- Can be used with external resultMap

MAPPERS in XML
Discriminator

MyBatis

- Works as **switch/case statement**

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin" />
  <result property="year" column="year" />
  <result property="make" column="make" />
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultMap="carResult" />
    <case value="2" resultMap="truckResult" />
    <case value="3" resultMap="vanResult" />
    <case value="4" resultMap="suvResult" />
  </discriminator>
</resultMap>
```

- Allows to:
  - switch class type
  - Remove properties
  - Add properties

```
<resultMap id="carResult" type="Car">
  <result property="doorCount" column="door_count" />
</resultMap>
```

will load ONLY doorCount property

```
<resultMap id="carResult" type="Car" extends="vehicleResult">
  <result property="doorCount" column="door_count" />
</resultMap>
```

will load doorCount, and properties from vehicleResult

MAPPERS in XML
Dynamic queries

- If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

- MyBatis uses **OGNL** (Object-Graph Navigation Language) - open-source **Expression Language** (EL) for Java. Used for getting and setting values. It also allows for dynamic execution of object's Java methods. It's known from frameworks:
  - WebWork and its successor Struts2
  - Tapestry (4 and earlier)
  - Spring Web Flow
  - Apache Click
  - …

- Basic statements: **if, choose (when, otherwise), trim (where, set), foreach**

MAPPERS in XML
Dynamic queries

- If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

- MyBatis uses **OGNL** (Object-Graph Navigation Language) - open-source **Expression Language** (EL) for Java. Used for getting and setting values. It also allows for dynamic execution of object's Java methods. It's known from frameworks:
  - WebWork and its successor Struts2
  - Tapestry (4 and earlier)
  - Spring Web Flow
  - Apache Click
  - …

- Basic statements: **if, choose (when, otherwise), trim (where, set), foreach**

MAPPERS in XML

If, Choose / Otherwise statements

MyBatis

- IF

```
<select id="findActiveBlogLike" parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <if test="title != null ">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

- Choose / Otherwise

```
<select id="findActiveBlogLike" parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <choose>
    <when test="title != null "> AND title like #{title}</when>
    <when test="author != null and author.name ! =null"> AND author_name like #{author.name} </when>
    <otherwise>AND featured = 1</otherwise>
  </choose>
</select>
```

MAPPERS in XML
Introduction to where statement

MyBatis

- What are possible outputs for the following query?

```
<select id="findActiveBlogLike" parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null ">
    state = #{state}
  </if>
  <if test="title != null ">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_namelike #{author.name}
  </if>
</select>
```

```
SELECT * FROM BLOG
WHERE
```

```
SELECT * FROM BLOG
WHERE
AND title like 'someTitle'
```

Will fail in some cases!

MAPPERS in XML
Where / Set

MyBatis

- WHERE statement simplifies notations, and stripes ANDs and ORs if needed:

```
<select id="findActiveBlogLike" parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null ">
      state = #{state}
    </if>
    <if test="title != null ">
      AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
      AND author_namelike #{author.name}
    </if>
  </where>
</select>
```

- Can be replaced by trim statement:

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
…
</trim>
```

- SET statement does the same thing for updates – it stripes comas at the end

MAPPERS in XML
For statement

MyBatis

- Often to to build IN conditions:

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list" open="(" separator="," close=")">
    #{item}
  </foreach>
</select>
```

- The element is smart in that it won't accidentally append extra separators.
- Can pass Lists and Arrays

QUERIES in XML
Insert, Update, Delete

MyBatis

- Another section are updatable queries – HOMEWORK

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
  insert into Author (id,username,password,email,bio)
  values (#{id},#{username},#{password},#{email},#{bio})
</insert>

<update id="updateAuthor" parameterType="domain.blog.Author">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

<delete id="deleteAuthor" parameterType=" int ">
  delete from Author where id = #{id}
</delete>
```

- Supports auto-generated keys by DB if DB and driver supports it
- Key generator – selectKey statement

MyBatis

@nnotation way

Queries
With annotations

- Since the very beginning, MyBatis has been an XML driven framework. The configuration is XML based, and the Mapped Statements are defined in XML. With MyBatis 3, there are new options available.
- Java Annotations are unfortunately limited in their expressiveness and flexibility. Despite a lot of time spent in investigation, design and trials, the most powerful MyBatis mappings simply cannot be built with Annotations – without getting ridiculous that is.

- Sample insert:

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")
@SelectKey(statement = "call next value for TestSequence", keyProperty = "nameId",
           before = true, resultType = int.class)
int insertTable3(Name name);
```

- Sample select:

```
@Insert("insert into table2 (name) values(#{name})")
@SelectKey(statement="call identity()", keyProperty="nameId", before=false, resultType=int.class)
int insertTable2(Name name);
```

Annotation way
Example

MyBatis

Provider way

Queries
With provider

- Last option to build queries is to specify SqlProvider:

```
@SelectProvider(type = BlogQueriesProvider.class, method = "findAllBlogs")
public List<Blog> findAllBlogs();
```

- Then in BlogQueriesProvider create method with name *findAllBlogs* that returns SQL:

```
public String findAllBlogs() {
        BEGIN();
        SELECT("b.id, b.name, b.description");
        FROM("app.blog b");
        ORDER_BY("b.name");
        return SQL();
    }
```

- MyBatis provides static builder methods in **org.apache.ibatis.jdbc.SqlBuilder** that helps ito generate SQL
- Results must be mapped by annotations or either XML mappers

Annotation way
Example

Other
Not mentioned (yet)

MyBatis

- Integration with:
  - Guice
  - Spring
- Extra tools:
  - Schema generation http://mybatis.org/generator/
  - Schema migration http://www.mybatis.org/migrations/ or Liquibase / Flyway
- Handlers
- ObjectFactory
- Plugins
- Cache – II level
- Statement execution methods
- SQL element for reuse

MyBatis
Sumarize

MyBatis

- Unlike ORM frameworks MyBatis does not map Java objects to database tables but methods to SQL statements.
- MyBatis lets you use all your database functionality like stored procedures, views, queries of any complexity and vendor proprietary features. It is often a good choice for **legacy or de-normalized databases** or when it is required to have **full control of SQL execution**.
- **It simplifies coding** compared to JDBC. SQL statements are executed with a single line. This saves time and prevents common mistakes like leaving a connection opened, coding a wrong data mapping, exceeding the limits of a result set or getting more than one result when just one was expected.
- MyBatis provides a mapping engine that maps SQL results to object trees in a declarative way.
- SQL statements can be built dynamically by using a custom dynamic language with XML-like syntax.
- MyBatis integrates with Spring Framework and Google Guice. This feature allows to build business code free of dependencies and even without any call to MyBatis API.
- MyBatis supports declarative data caching. It integrates with: OSCache, EHcache, Hazelcast and Memcached and supports custom integration with other cache tools.

rule FINANCIAL

The Sector Specialists

ANY QUESTIONS?