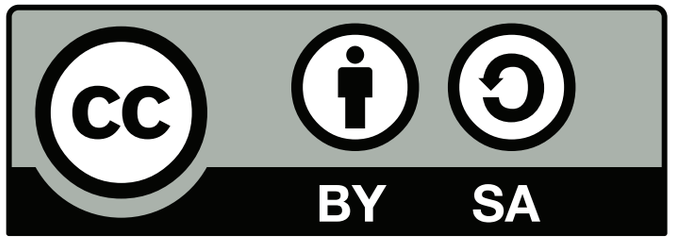
Tecnologia e Applicazioni Internet 2008/9

Lezione 9 - Persistenza

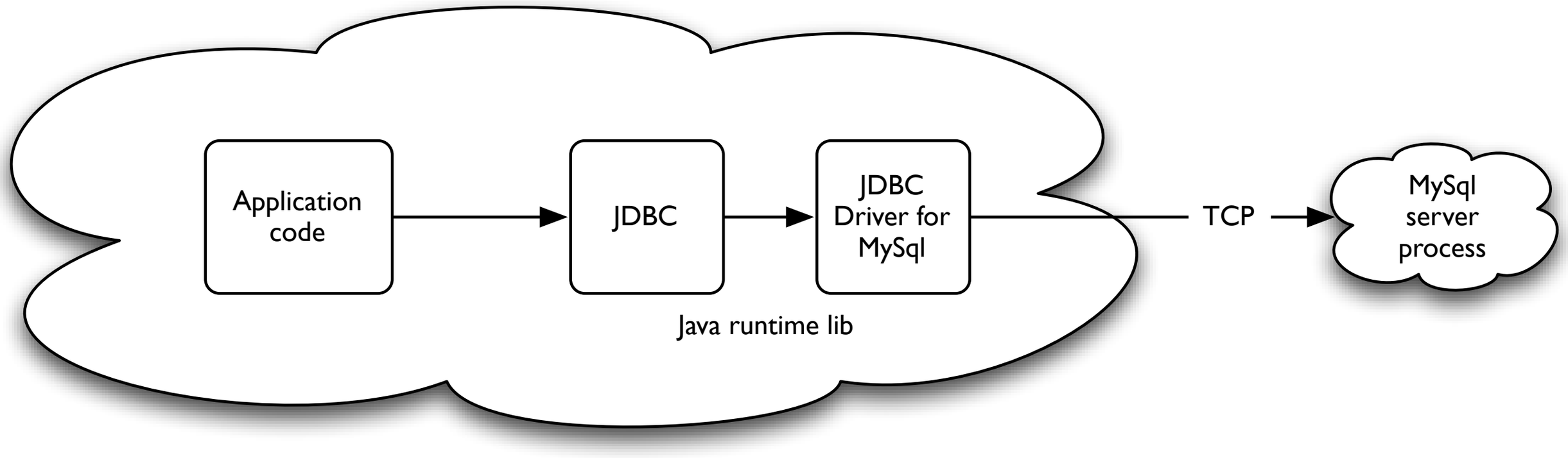
Matteo Vaccari
http://matteo.vaccari.name/
matteo.vaccari@uninsubria.it



Perché usare un DB relazionale?

- Per l'accesso concorrente ai dati (e svincolare il codice applicativo dalla concorrenza)
- Per estrarre i dati in maniera veloce
- Per fare fronte a nuovi requisiti tramite una semplice riconfigurazione dello schema (cf. usare il filesystem)

Java and JDBC



Get a JDBC connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnector {
    // Should be loaded from external configuration
    final String USERNAME = "myuser";
    final String PASSWORD = "mypassword";
    final String URL = "jdbc:mysql://localhost/mydatabase";
    final String DRIVER = "com.mysql.jdbc.Driver";

    public Connection getConnection() throws ClassNotFoundException, SQLException {
        // load JDBC driver
        Class.forName(DRIVER);

        // create connection
        return DriverManager.getConnection(URL, USERNAME, PASSWORD);
    }
}
```

Execute sql code

```
Statement statement = connection.createStatement();

String sql = "INSERT INTO my_table (col_string) VALUES('a string')";

statement.executeUpdate(sql);
```

Use a *prepared statement*

```
String sql = "INSERT INTO my_table (col_string) VALUES (?)";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setString(1, "row "+i);
statement.executeUpdate();
```

... and close the statement

```
PreparedStatement statement;
try {
    String sql = "INSERT INTO my_table (col_string) VALUES (?)";
    statement = connection.prepareStatement(sql);
    statement.setString(1, "row "+i);
    statement.executeUpdate();
} finally {
    if (null != statement) {
        try {
            statement.close();
        } catch(Exception ignored) {}
    }
}
```

Note

- *statement.finalize()* chiuderebbe lo statement, ma viene chiamato dal garbage collector non si sa quando
- Bisogna chiudere esplicitamente lo statement, altrimenti se abbiamo molte operazioni concorrente alcune falliranno
- Bisogna ignorare le eccezioni in *statement.close()*, altrimenti *oscureranno* l'eventuale eccezione lanciata da *statement.executeUpdate()*

Reading data from a table

```
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");

while (resultSet.next()) {
    String s = resultSet.getString("col_string");
}
```

... and close objects

```
Statement statement;
ResultSet resultSet;
try {
    statement = connection.createStatement();
    resultSet = statement.executeQuery("SELECT * FROM my_table");

    while (resultSet.next()) {
        String s = resultSet.getString("col_string");
    }
} finally {
    if (null != resultSet) {
        try {
            resultSet.close();
        } catch(Exception ignored) {}
    }
    if (null != statement) {
        try {
            statement.close();
        } catch(Exception ignored) {}
    }
}
```

Usare uno script per generare il database

- Crea due database, uno per unit test e uno per sviluppo
- Però prima li cancella se esistono
- Carica lo schema dei dati
- Crea un utente applicativo e gli dà i diritti

Usare uno script per generare il database, perchè?

- Bisogna sempre automatizzare tutto
- Mette i colleghi in grado di partire velocemente
- Cristallizza le informazioni necessarie per installare l'applicazione
- Se ho lo script, modificare lo schema costa poco

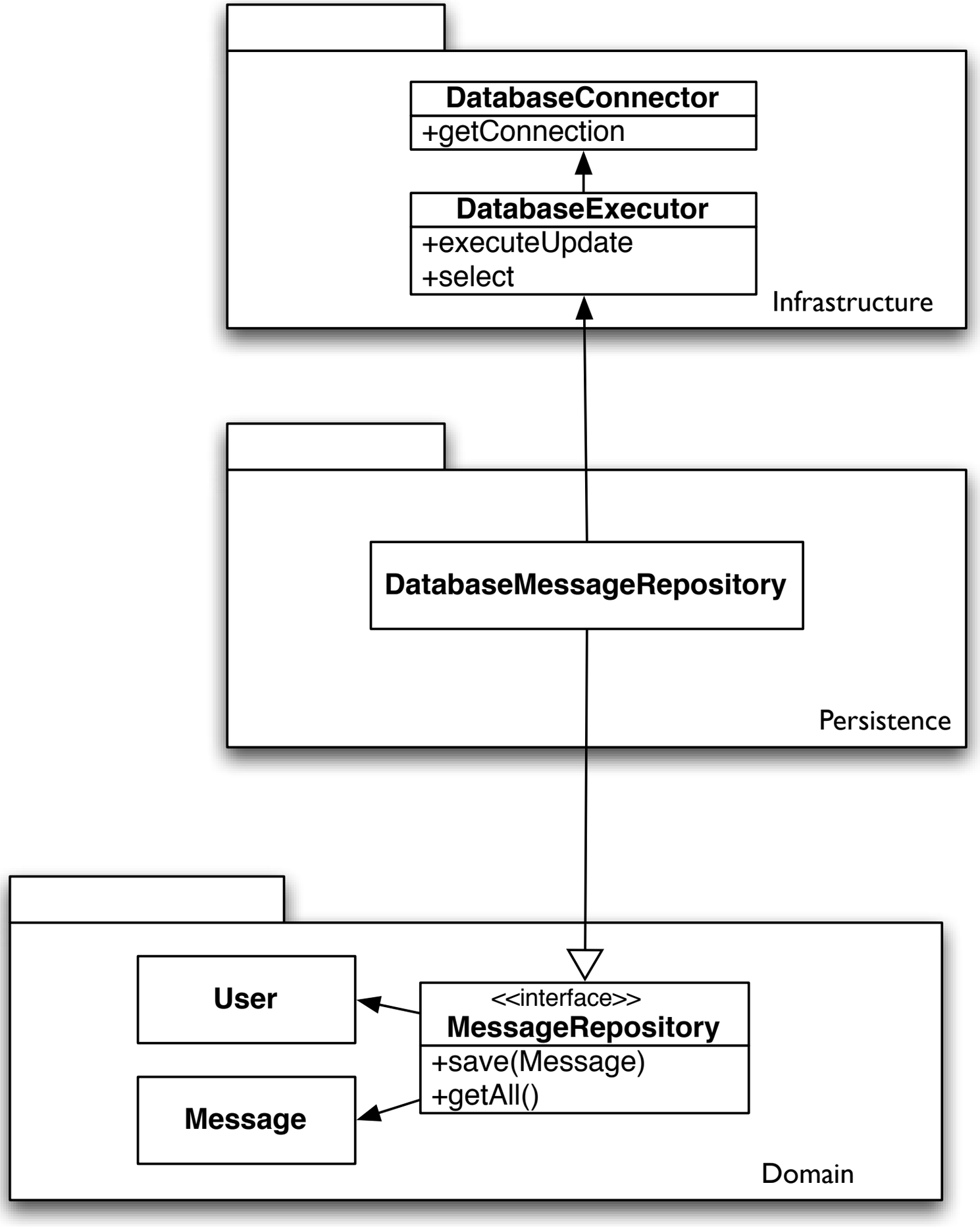
```
# define key information
src=sql                     # sql sources directory
dbname=tai_chat             # name of development db
dbname_test=tai_chat_test # name of test db
dbuser=tai_chat             # name of application user
dbpassword=tai_chat         # password of application user

# ask mysql root password
read -s -p "mysql root password? (type return for no password) " MYSQL_ROOT_PASSWORD
if [ "$MYSQL_ROOT_PASSWORD" != "" ]; then
    MYSQL_ROOT_PASSWORD=-p$MYSQL_ROOT_PASSWORD
fi

# drop and create databases
mysqladmin -uroot $MYSQL_ROOT_PASSWORD drop $dbname
mysqladmin -uroot $MYSQL_ROOT_PASSWORD --force drop $dbname_test
mysqladmin -uroot $MYSQL_ROOT_PASSWORD create $dbname
mysqladmin -uroot $MYSQL_ROOT_PASSWORD create $dbname_test

# create application user and give grants
echo "grant all on $dbname.* to '$dbuser'@localhost identified by '$dbpassword';" \
     | mysql -uroot $MYSQL_ROOT_PASSWORD $dbname
echo "grant all on $dbname_test.* to '$dbuser'@localhost identified by '$dbpassword';" \
      | mysql -uroot $MYSQL_ROOT_PASSWORD $dbname_test

# load schema
cat $src/???_*.sql | mysql -u$dbuser -p$dbpassword $dbname
cat $src/???_*.sql | mysql -u$dbuser -p$dbpassword $dbname_test
```


DatabaseConnector
+getConnection
DatabaseExecutor
+executeUpdate
+select
Infrastructure
DatabaseMessageRepository
Persistence
User
<<interface>>
MessageRepository
+save(Message)
+getAll()
Message
Domain

The connection life-cycle

```
public class ChatServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) {
        // build a network of objects
        DatabaseConnector connector = new DatabaseConnector(
            "tai_chat", "tai_chat", "jdbc:mysql://localhost/tai_chat_test",
            "com.mysql.jdbc.Driver");

        // executor will get a connection when needed
        DatabaseExecutor executor = new DatabaseExecutor(connector);
        DatabaseMessageRepository repository = new DatabaseMessageRepository(executor);

        try {
            // respond to user request
            // ...
        } finally {
            // close database connection
            executor.closeConnection();
        }
    }
}
```

One connection per request

The executor

```
public void execute(String sql, Object ... params) {
    Connection connection = getConnection();
    PreparedStatement statement = null;
    try {
        statement = connection.prepareStatement(sql);
        for (int i=0; i<params.length; i++) {
            statement.setObject(i+1, params[i]);
        }
        statement.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        safelyClose(statement);
    }
}

// usage example
String sql = "insert into my_table (foo, bar) values (?, ?)";
executor.execute(sql, "a string", 123);
```

Mapper objects

```
// map a resultSet row to a domain object
public interface Mapper<T> {
    public T map(ResultSet resultSet);
}

public class MessageMapper implements Mapper<Message> {

    public Message map(ResultSet resultSet) {
        try {
            return new Message(resultSet.getString("text"));
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
// in class DatabaseExecutor
public <T> List<T> select(String sql, Mapper<T> mapper) {
    Connection connection = getConnection();
    Statement statement = null;
    ResultSet resultSet = null;
    try {
        statement = connection.createStatement();
        resultSet = statement.executeQuery(sql);
        List<T> result = new ArrayList<T>();
        while (resultSet.next()) {
            T object = mapper.map(resultSet);
            result.add(object);
        }
        return result;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        safelyClose(resultSet);
        safelyClose(statement);
    }
}

// in class MessageRepository
public List<Message> getAllMessages() {
    return executor.select("select * from messages", new MessageMapper());
}
```