Complex Systems Theory and Practice

Lecture 10

Piotr Syga

2021-12-14

Supported dialects and configuration

- SQLite
- MySQL, MariaDB
- Oracle
- PostgreSQL
- other

Database configuration

Let us assume that we create a project `ddb` in a virtual environment (`virtualenv`). We edit `settings.py` (by default ~/ddb/ddb). In `DATABASES` field, we edit the document `default` (or we add another one).
Important fields: `ENGINE` (`django.db.backends.*` sqlite by default, we can change to `mariadb`), `NAME`, `USER`, `PASSWORD`, `HOST`, `PORT`).
After that we migrate the data or create structures: `python manage.py makemigrations`, `python manage.py migrate` and run the server `python manage.py runserver <host>:<port>`.

Models as tables

We create `models.py` in the folder

```
from django.db import models

  class Student(models.Model):
        IndexNo = models.IntegerField()
        FirstName = models.CharField(max_length=90)
        LastName = models.CharField(max_length=90)
        Department = models.IntegerField()
        Major = models.CharField(max_length=120)
        Notes = models.CharField(max_length=256)
```

Web form

We create `forms.py`

```
from django import forms
from .models import Student

  class StudentForm(forms.ModelForm):
        class Meta:
            model = Student
            fields = ['IndexNo', 'FirstName',
            'LastName', 'Department',
            'Major', 'Notes']
```

Creating of web view

We start with creating a template `templates/create_view.html`, and then declare the form's actions `views.py`

```
from django.shortcuts import render
from .models import Student
from .forms import StudentForm

def create_view(request):
    context ={}
    form = StudentForm(request.POST or None)
    if form.is_valid():
        form.save()

    context['form']= form
    return render(request, "create_view.html", context)
```

Update

```
from django.shortcuts import (get_object_or_404, render,
                              HttpResponseRedirect)
from .models import Student
from .forms import StudentForm

def detail_view(request, id):
    context ={}
    context["data"] = Student.objects.get(id = id)
    return render(request, "<StudentsWindow>.html", context)
def update_view(request, id):
    context ={}
    obj = get_object_or_404(Student, id = id)
    form = Student(request.POST or None, instance = obj)
    if form.is_valid():
        form.save()
        return HttpResponseRedirect("/"+id)
    context["form"] = form
    return render(request, "<updater>.html", context)
```

Inline operations

Using Anacondy or a similar environment we can use scripts or inline operations. Let us assume that there is a Courses class that represents an instance of a group, with `signedin` filed that denotes the students in the class and `dates` with dates of the classes.

```
> from CS.models import Student
> JD= Student.objects.create(IndexNo='222222', FirstName="John",
LastName="Doe")
> AN= Student.objects.create(IndexNo='222224', FirstName="John",
LastName="Smith")
> Courses.signedin.add(JD, JS)
> Course = Courses.objects.all()
> TCourse = Course.filter(dates=datetime.date.today())
> Johns = Student.objects.get(FirstName__contains='John')
```

Transactions

atomic() and ATOMIC_REQUESTS

By default all operations are sequential, that might be changes by `atomic()` context menager. During the configuration of `settings.py` we can set `ATOMIC_REQUESTS` and `AUTOCOMMIT` fields to `true` and `false`, respectively, such that the default way will be full usage of transactions. To use sequential operations i fully transactional database we use `non_atomic_requests(using=None)` context manager.

Transactions – schemes

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    -- atomic operations

from django.db import transaction

@transaction.non_atomic_requests
def viewfunc(request):
    -- operations that are sequential
```

Introduction

Main applications

- web applications
- Android
- Hibernate support
- client–server application via TCP

Main properties

- opensource Java-based engine with JDBC API support
- PostgreSQL ODBC driver, supports also Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle
- allows multiple concurrent connections with row-level locking
- optimized (cost based)
- Encrypted database (AES), SHA-256 password encryption
- Transactions and distributed transactions (2PC)
- procedures, functions, triggers, foreign keys, indexes, prepared statements, window functions
- `EXPLAIN PLAN` statement
- support for users and roles

TCP Server

```
import org.h2.tools.Server;
/* imports
*/
Server svr = Server.createTcpServer(args).start();/*
 TCP server start
*/
/* commands
*/
svr.stop(); /* closing the server */
```

Connection

We use org.h2.Driver

```
import java.sql.*;
public class Test {
    public static void main(String[] a)
            throws Exception {
        Connection conn = \
DriverManager.getConnection("jdbc:h2:~/db", "user", "pass");
        /* "jdbc:h2:~/db;CIPHER=AES" for encryption,
"jdbc:h2:~/db;IFEXISTS=TRUE"
so no new db is created */
/*
JDBC commands, cf. the last lecture
*/
        conn.close();
    }
}
```

Closing the connection

- By default the database is closed when the last connection is closed.
- In order to prevent frequent opening and closing the database, we may delay the closing with `SET DB_CLOSE_DELAY <seconds>`, and with -1 value the database is never closed automatically.
- Delay may be set also on a connection step if the connecting user is an administrator `jdbc:h2:~/db;DB_CLOSE_DELAY=10`.
- Similarly, when the last Java VM is closed, the database is closed by default, however we may change it `jdbc:h2:~/db;DB_CLOSE_ON_EXIT=FALSE`

Triggers

H2 fully supports Java triggers (and have to implement `org.h2.api.Trigger` interface), following the JDBC-inspired syntax

```
import org.h2.api.Trigger;
public class TriggerSample implements Trigger {
    public void init(Connection conn, String schemaName, String triggerName,
            String tableName, boolean before, int type) {
        // initialize the trigger object is necessary
    }

    public void fire(Connection conn,
            Object[] oldRow, Object[] newRow)
            throws SQLException {
        // the trigger is fired
    }

    public void close() {
        // the database is closed
    }

    public void remove() {
        // the trigger was dropped
    }

}

CREATE TRIGGER Stud_INS AFTER INSERT ON STUDENTS
    FOR EACH ROW CALL "org.h2.samples.TriggerSample"
```

Backup & Restore

- console:
  `java -cp h2*.jar org.h2.tools.Backup`
  (For more flags and console tools use `-?` option)
- script tool:
  `java org.h2.tools.Script -url jdbc:h2: /test -user <user_name> -script test.zip -options compression zip`
  (Note: this command may be run from SQL level by `SCRIPT`)
- On-line backup:
  `BACKUP TO 'backup.zip'`

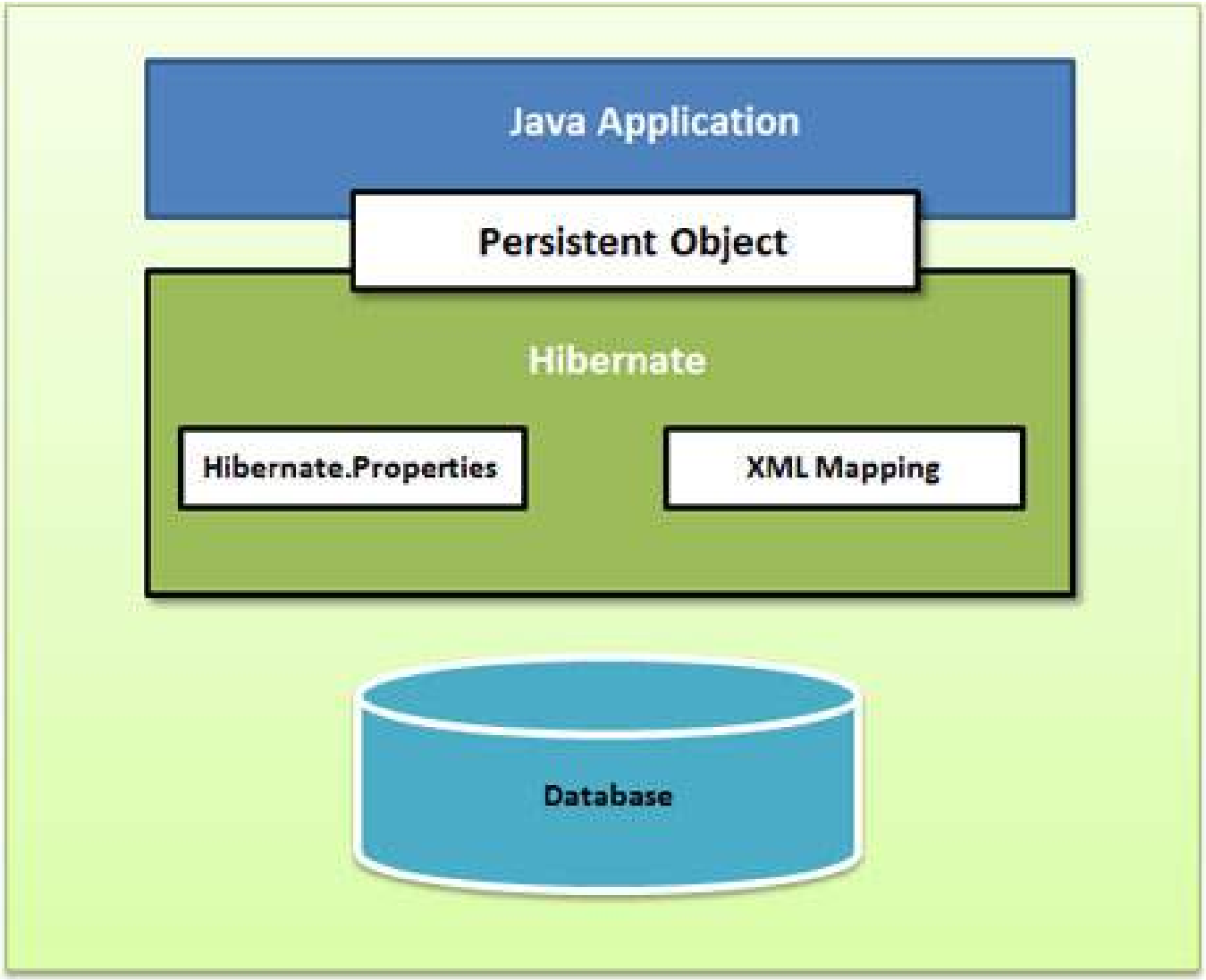
Backup & Restore

- console:
  `java -cp h2*.jar org.h2.tools.Restore`
  (If the backup is corrupted or incomplete one may try to use the recovery tool: `org.h2.tools.Recover`)
- script tool:
  `java org.h2.tools.RunScript -url jdbc:h2: /test -user <user_name> -script test.zip -options compression zip`

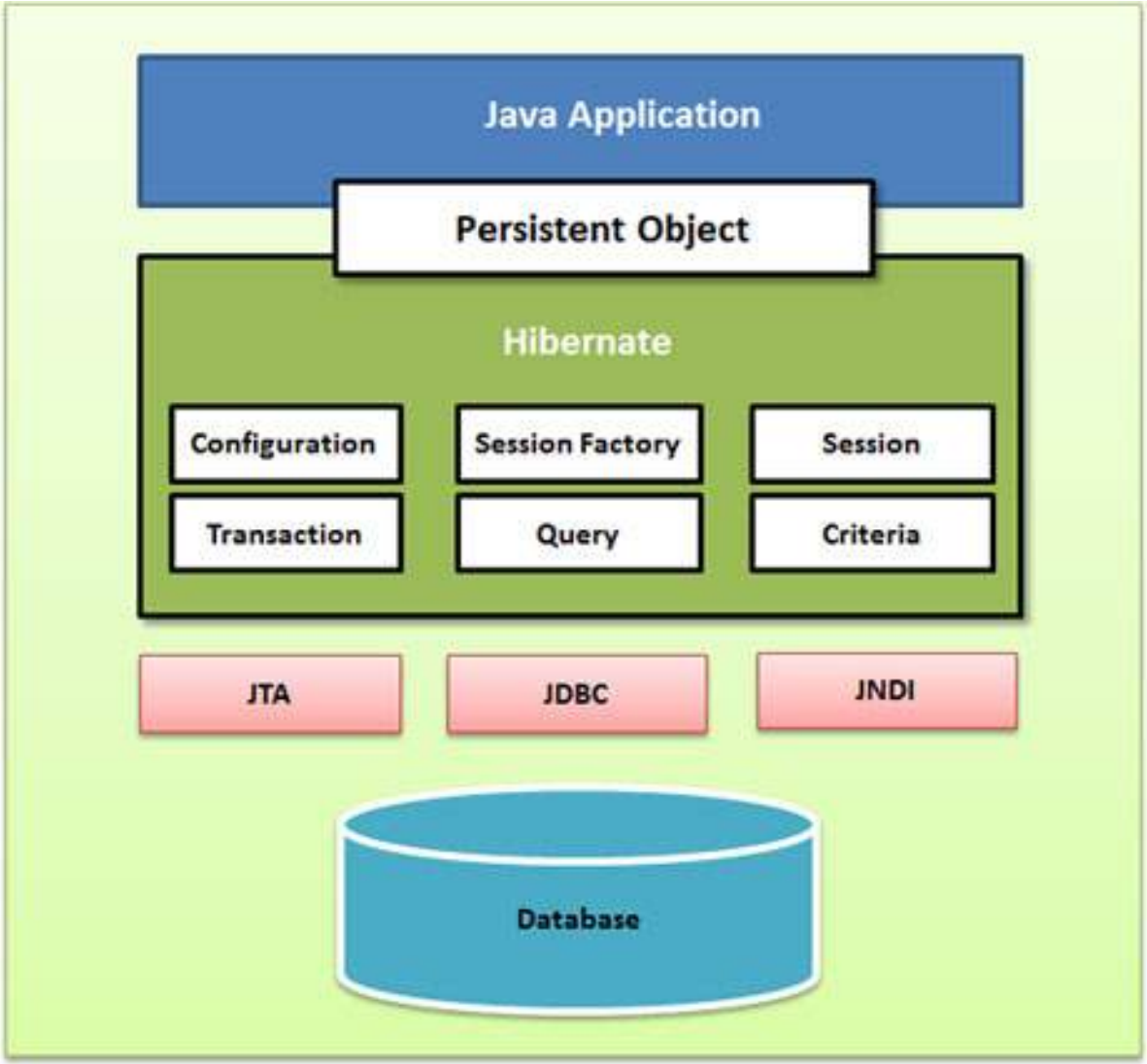
Hibernate ORM

- a high-performance Object-Relational framework
- maps the Java classes into SQL datatypes and queries
- includes query and retrieval mechanisms

Hibernate ORM



Hibernate ORM



Hibernate ORM

Configuration file:

- Hibernate's configuration file is stored using a standard XML.
- The main property is `<hibernate-configuration>` with `<session-factory>` tag following it as the main embedded branch.
- The main properties of a session include:
  - hibernate.dialect, e.g., `org.hibernate.dialect.MySQLDialect`
  - hibernate.connection.driver_class, e.g., `com.mysql.jdbc.Drive`
  - hibernate.connection.url, e.g., `jdbc:mysql://localhost/db`
  - hibernate.connection.username and hibernate.connection.password
- Moreover, we should specify a mapping resource.

Hibernate ORM

For running queries, one would use a standard SQL syntax enriched with Java objects, e.g.

```
String hql = "SELECT AVG(S.mark), S.fName, S.lName FROM
Students S " +
"GROUP BY S.ID"+"ORDER BY S.lName";
Query query = session.createQuery(hql);
List results = query.list();
```

We can also delay the evaluation of the parameters:

```
String hql = "UPDATE Students set mark = :grade " +
"WHERE id = :sid";
Query query = session.createQuery(hql);
query.setParameter("grade", 5);
query.setParameter("sid", 150000);
int result = query.executeUpdate();
System.out.println("Rows affected:  " + result);
```

Hibernate ORM

- Additional conditions may be added by using `Restrictions` in `Criterias`.
- We start with creating a criteria in a given session `Criteria cr = session.createCriteria(Students.class);`.
- We can add various restrictions including
  - equality: `cr.add(Restrictions.eq("course", 'Machine Learning'))`
  - inequality: `cr.add(Restrictions.gt("mark", 3.5));`
  - substrings: `cr.add(Restrictions.like("fName", "jo%"));` (and case sensitive `cr.add(Restrictions.ilike("fName", "Jo%"));`)
  - nulls: `cr.add(Restrictions.isNotNull("course"));`
  - logical operations: `cr.add( Restrictions.or( R1, R2) );` and `cr.add( Restrictions.and( R1, R2) );`
- Criteria may include also sorting (`cr.addOrder(Order.desc("lName"));`) and projections (`cr.setProjection(Projections.avg("mark"));`)

Introduction

- Cross- platform Framework (open-source)
- ORM with connectors to multiple SQL dialects
- Python-native with (among others) Django and Flask support

Supported dialects

- MySQL, MariaDB
- PostgreSQL
- OracleDB
- Microsoft SQL Server
- SQLite
- externals (Redshift, CreateDB, Firebird, . . .)

Connecting to the database

- import: `from sqlalchemy import create_engine`
- db object creation: `engine = create_engine("dialect[+driver]://user:pass@host/dbname", echo = True)`
  (we can use mock or predefined data, cf. `create_mock_engine`, `engine_from_config`)

Connecting to the database

Selected methods from Engine

- `sqlalchemy.engine.Engine.connect(close_with_result=False)` – returns a new handler to the connection
- `sqlalchemy.engine.Engine.dispose()` – rejects `sqlalchemy.pool.Pool` of the closed connections
- `sqlalchemy.engine.Engine.has_table(table_name, schema=None)`– checking for the existing tables
- `sqlalchemy.engine.Engine.execute(stmt, *multiparams, **params)` – executing SQL queries (**Note**: currently moved to `Connection.*`, kept only for compability)
- `sqlalchemy.engine.Engine.execution_options(**opt)` – creates a new `sqlalchemy.engine.Connection` object
- `sqlalchemy.engine.Engine.begin(close_with_result=False)` – starting a new transaction

Mapping

```
from sqlalchemy import Column, Integer, String
class Students(Base):
    __tablename__ = 'Students'
    indexNo = Column(Integer, primary_key = True)
    name = Column(String)
    lastName = Column(String)
    semester = Column(Integer)
    email = Column(String)
```

Queries

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()
result = session.query(Students).filter(Students.semester>2)

for row in result:
   print ("Index number:", row.indexNO, "Name: ",row.name,
 "Last Name:",row.lastName, "Semester":, row.semester,
 "Email:",row.email)
```

Update

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()
result = session.query(Students).filter(Students.semester = 7).
update({Students.name:"inż. "+Students.name,
 Students.semester: 0}, synchronize_session = False)
```

Delete

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
z = session.query(Students).get(150931)
session.delete(z)

session.query(Students).filter_by(lastName = 'Doe').
count()
```

Joins

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()

for s, m in session.query(Students, Major).
filter(Student.indexNo == Major.student).all():
    print (...)
```