

COMP S380F Lecture 8: Data Access Object (DAO), Hibernate, Spring Data JPA

Dr. Keith Lee

School of Science and Technology

Hong Kong Metropolitan University

Overview of this lecture

- Review on JDBC
- Object-Relational Mapping (ORM) tool: Hibernate
 - Entity classes
- Data Access Object (DAO) pattern
 - Repository interface & implementation
- Webapp example: **HelloSpringHibernate**
- JPA vs. Hibernate
- Spring Data JPA with Hibernate
 - Webapp example: **HelloSpringDataJPA**
- Automatic JPA repository with Spring Data JPA
- Defining repository method
 - By name convention
 - By @Query annotation
 - By implementation

We are generating vast amount of data!!

Remote patient
monitoring

Healthcare

Product sensors

Manufacturing

...

Social media

Retail

books, music,
videos, etc.

Digitization of Artefacts

Real time location
data

Location-Based Services

We are generating vast amount of data!! (cont')

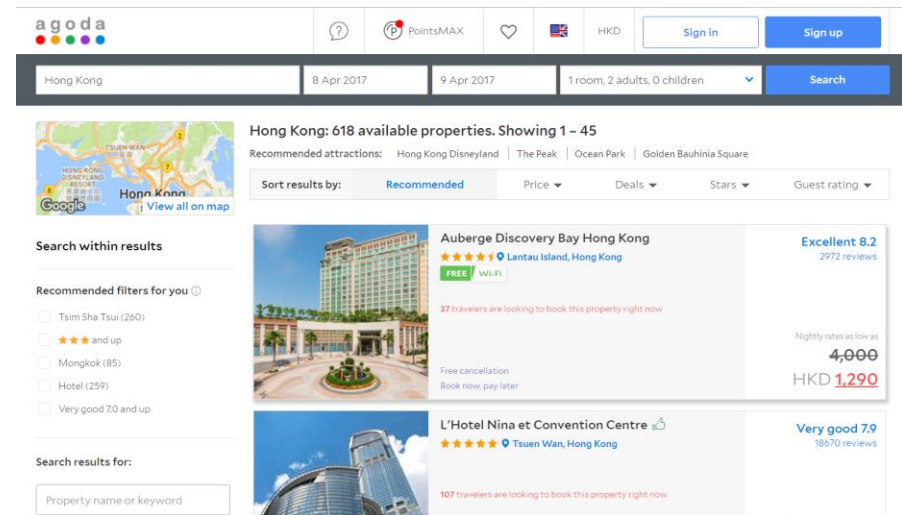
- Air Bus A380:
 - Generate 10 TB of data every 30 min
- Twitter:
 - Generate ~12 TB of data per day
- Facebook:
 - Facebook data grows by over 500 TB daily
- New York Stock:
 - Exchange 1 TB of data everyday

Challenge

- How do we store and access this data over the web?

E-commerce website

- Data operations are mainly transactions (Reads and Writes)
- Operations are mostly online
- Response time should be quick but it is important to maintain **security and reliability of transactions**
- ACID properties are important



Challenge

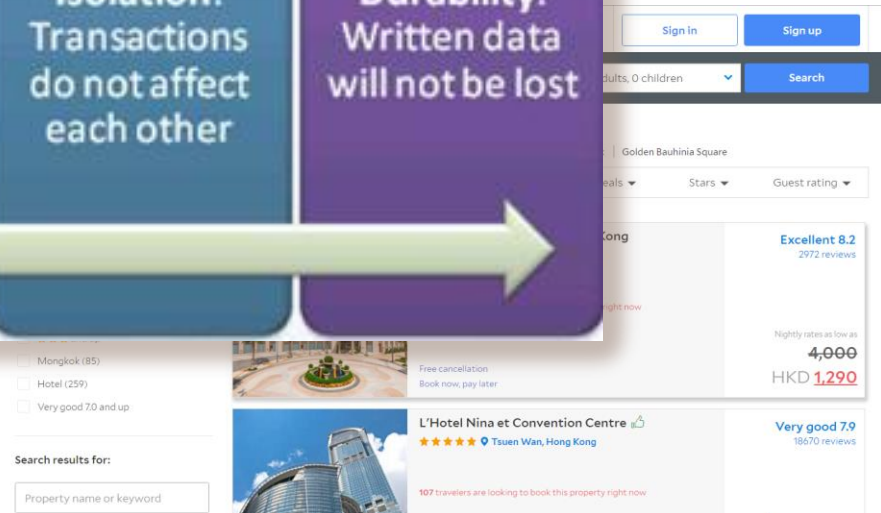
- How do we store and access this data over the web?

E-commerce

- Data op
- Operati
- Respon
- but it is
- security**
- transac**



- ACID properties are important



Challenge (cont')

- How do we store and access this data over the web?



Image serving website

- Data operations are mainly fetching large files (Reads)
- ACID requirements can be **relaxed**
 - E.g., It is hard to maintain *consistency* due to lack of direct control of data from users, so we may relax the requirement to only ensure consistent outcomes from the data.
 - E.g., Locking shared data may result in denial of service, so the level of *isolation* may not be as high as traditional system.
- Operations are mainly online
- High bandwidth requirement

Challenge (cont')

- How do we store and access this data over the web?

Search engine

- Data operations are mainly reading index files for answering queries (Reads)
- ACID requirements can be relaxed
- Index compilation is performed offline due to the large size of source data (the entire Web)
- Response time must be as fast as possible



Data persistence

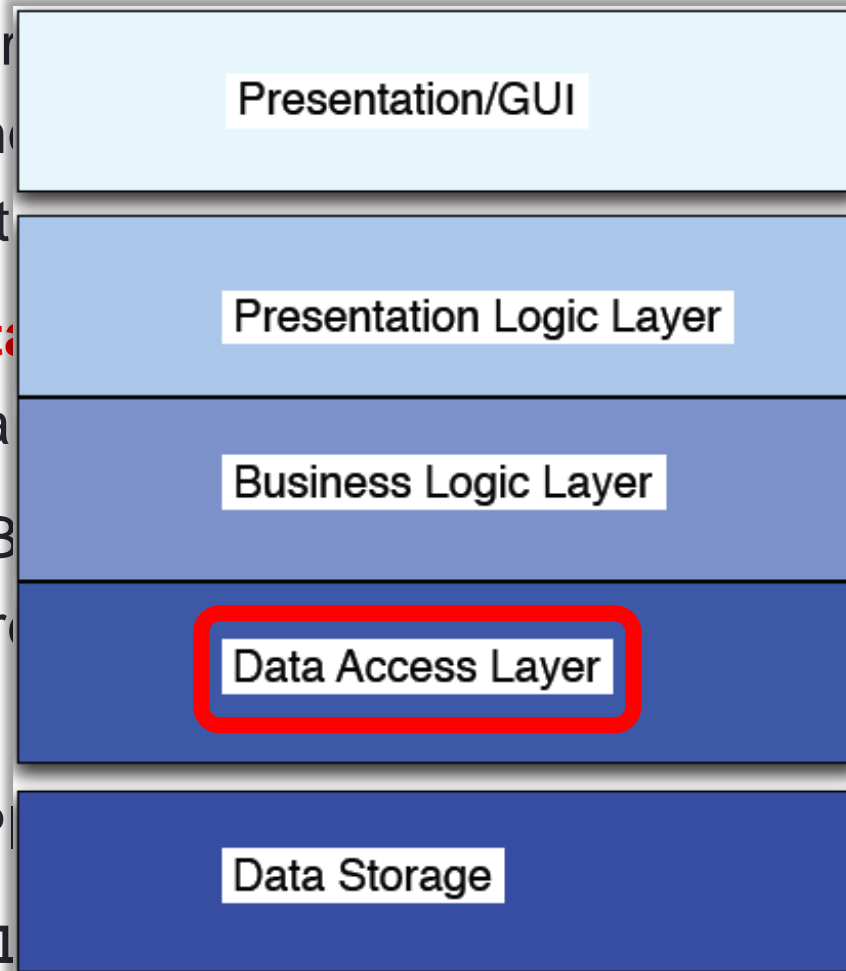
- When we talk about persistence in Java, we normally mean storing data in a relational database using SQL.
- Relational database technology is a common denominator for many disparate systems and technology platforms.
- Relational database provides a way of **sharing data** across different applications or technologies that form part of the same application.
- The relational data model is often the common enterprise-wide presentation of business entities.

Data persistence (cont')

- When you work with a relational database in a Java application, the Java code issues SQL statements to the database via the JDBC API.
- The **Java Database Connectivity (JDBC) API** provides universal data access from the Java programming language.
- Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files.
- The JDBC API is comprised of two packages:
 - `java.sql`
 - `javax.sql`

Data persistence (cont')

- When you work with a Java application, the application interacts with the database via the layers shown in the diagram.
- The **Java Data Objects** (JDO) provides a universal data access API in Java.
- Using the JDBC API, you can access any data source, from relational databases to spreadsheets and flat files.
- The JDBC API is defined by the `java.sql` and `javax.sql` packages.

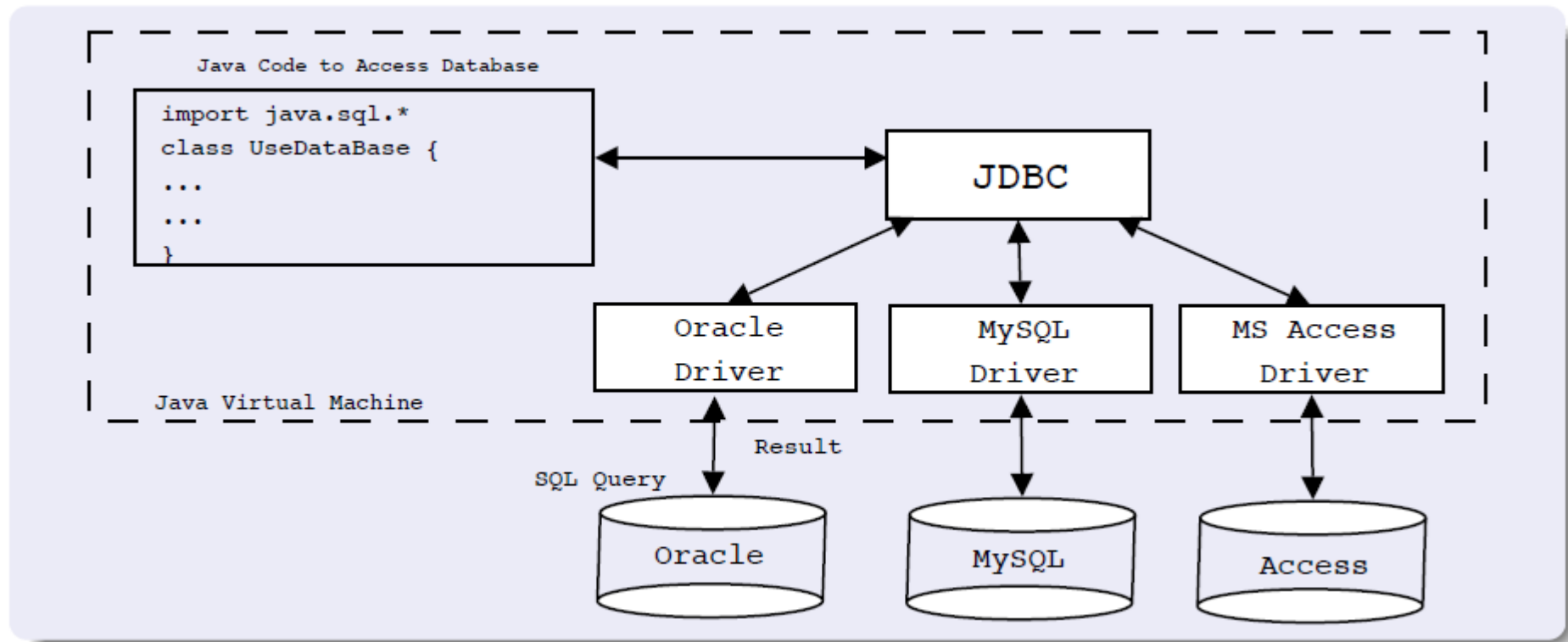


JDBC: Accessing DB from an application

JDBC (Java Database Connectivity) API:

- JDBC is a package (`java.sql.*`) of Java classes that provides access to relational database management system (RDBMS, e.g., Oracle, PostgreSQL) via SQL.
- JDBC offers simplicity (easy to use) and database independence
 - Developers do not need to worry about differences in vendor-specific database connection instructions
- A vendor would have to produce JDBC drivers and build their proprietary connection management code beneath a common Java API
- A **JDBC driver** is a segment of code designed to enable access to a *particular kind* of database.

JDBC: Accessing DB from an application (cont')



- Java classes in the JDBC package (`java.sql.*`) are interfaces.
- RDBMS vendors implement the interfaces to fit with their own products.

JDBC concepts

- When developers use JDBC, they construct SQL statements that can be executed. E.g.,

SELECT name FROM employee WHERE age = ?

- The above SQL is combined with local data structures so that regular Java objects can be mapped to the bindings in the string.

➤ E.g., an **Integer object with the value of 42** can be mapped:

SELECT name FROM employee WHERE age = 42

- The results of execution, if any, are combined in a set returned to the caller. E.g., the above SQL may return:

name

Peter
Mary
John



We can browse this result set as necessary.

JDBC interfaces

- `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

- 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 */
}
```


Object-Relational Mapping (ORM)

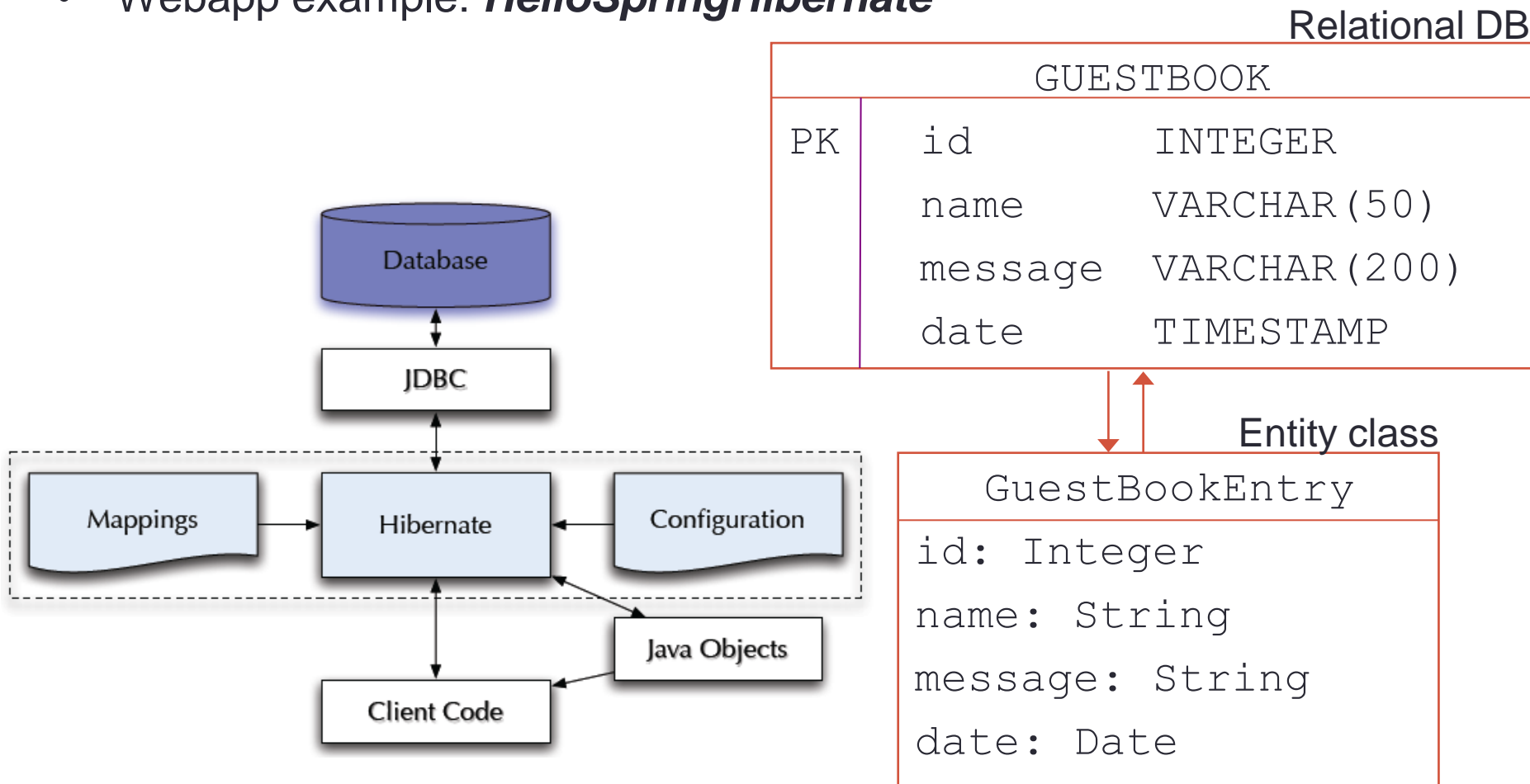
- JDBC is a primitive way for applications to talk to databases.
- **Object-relational mapping (ORM)** aims to give you automated (and transparent) persistence of objects in Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database.
- There are many tools / frameworks in this area:
 - JDO (Java Data Objects): <http://db.apache.org/jdo/>
 - **JPA (Java Persistence API):**
<https://www.oracle.com/java/technologies/persistence-jsp.html>
 - EclipseLink: <https://www.eclipse.org/eclipselink/>
 - **Hibernate:** <http://hibernate.org/>
- In this course, we will use Hibernate, which is one of the popular ORM tools (e.g., WildFly has built-in Hibernate support).

Features of ORM tools

- Object to Relational mapping using ordinary JavaBeans (**Entity class**)
- **Lazy loading**:
 - As object graphs become more complex (e.g., one-to-many, many-to-many mappings), you do not want to fetch entire relationships immediately.
 - Lazy loading allows you to grab data only as it is needed.
 - E.g., A **Teacher** has many **Courses** to teach (one to many).
 - The **Teacher** object is fetched, and the related **Course** objects are fetched only when needed.
- **Eager fetching**:
 - Opposite of lazy loading.
 - Eager fetching allows you to grab an entire object graph in one query, and thus saving you from costly round-trips in some cases.
- **Cascading**: Changes to one table result in changes to other tables.

Entity class

- An **entity class** maps to a database table for manipulating its records.
- It is often a JavaBean and is marked with **field annotations**.
- Webapp example: ***HelloSpringHibernate***



Entity class: Field annotations

GuestBookEntry.java

```
@Entity  
@Table(name = "guestbook")  
public class GuestBookEntry implements Serializable {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    private String name;  
    private String message;  
  
    @Temporal(TemporalType.TIMESTAMP)  
    private Date date;  
  
    // getters and setters for id, name, message, date  
}
```

- **@Entity** specifies an entity class.
- By default, the table name equals the class name.
 - Use **@Table** on the class if the table and class have different names.

Entity class: Field annotations (cont')

@Entity
@Table(name = "guestbook")
 public class GuestBookEntry implements Serializable {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Integer id;

 private String name;
 private String message;

 @Temporal(TemporalType.TIMESTAMP)
 private Date date;

 // getters and setters for id, name, message, date
 }

GuestBookEntry.java

For converting between
 java.sql.Timestamp &
 java.util.Date

Can have value DATE,
 TIME, or TIMESTAMP.

- By default, the table column name equals the property name.
 - Use **@Column** if they are different:


```
@Column(name = "employees_number")  
private Integer empINumber;
```
- **@Id** specifies the primary key of the table.
- **@GeneratedValue** specifies that property value will be automatically generated. **strategy** specifies how the primary key is generated.

Configuration for using Hibernate with Spring

Maven dependencies in **pom.xml**

- In addition to the dependencies for Derby DB, spring-jdbc, ...

```
<properties>
  ...
  <spring.framework.version>5.3.16</spring.framework.version>
  <spring.security.version>5.6.2</spring.security.version>
  <hibernate.version>5.6.5.Final</hibernate.version>
</properties>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>${spring.framework.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${spring.framework.version}</version>
</dependency>
```

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>${hibernate.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>${hibernate.version}</version>
</dependency>
```

Configuration for using Hibernate with Spring

Dispatcher's Spring application context in **dispatcher-servlet.xml**

```

<beans ...
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="...
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
">
  <context:component-scan
    base-package="hkmu.comps380f.controller, hkmu.comps380f.dao" />
  ...

  <!-- Hibernate 5 SessionFactory Bean -->
  <bean id="hibernate5SessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan">
      <list>
        <value>hkmu.comps380f.model</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
        <prop key="hibernate.show_sql">true</prop>
      </props>
    </property>
  </bean>

```

dispatcher-servlet.xml

DAO pattern
(to be explained)

Spring bean **dataSource** is
defined in the root Spring
application context (for
Spring Security)

Entity classes are here

Configuration for using Hibernate with Spring

Dispatcher's Spring application context in **dispatcher-servlet.xml**

dispatcher-servlet.xml

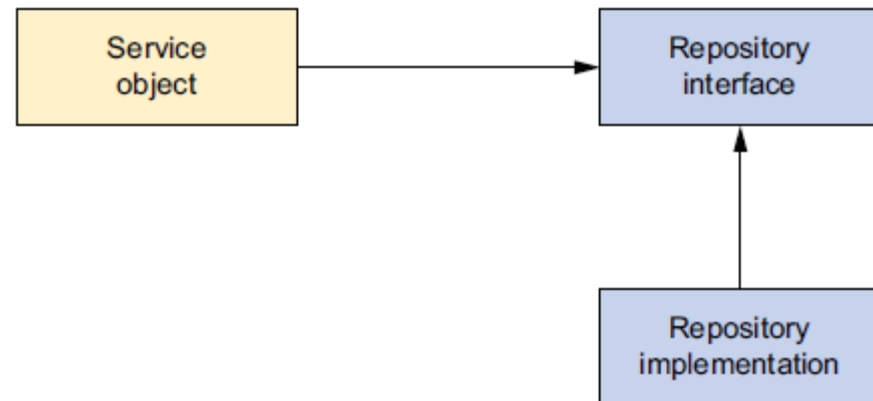
```
<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="transactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="hibernate5SessionFactory" />
</bean>
</beans>
```

- **transactionManager** is the Spring's transaction manager bean.
- Spring's **<tx:annotation-driven>** enables annotation-driven transaction management (We will use the annotation **@Transactional**).
- Hibernate **Session** object provides basic data-access functionality such as the ability to save, update, delete, and load entity objects from the database. (~ DB connection)
- Hibernate **session factory** is responsible for opening, closing and managing Hibernate **Sessions**.

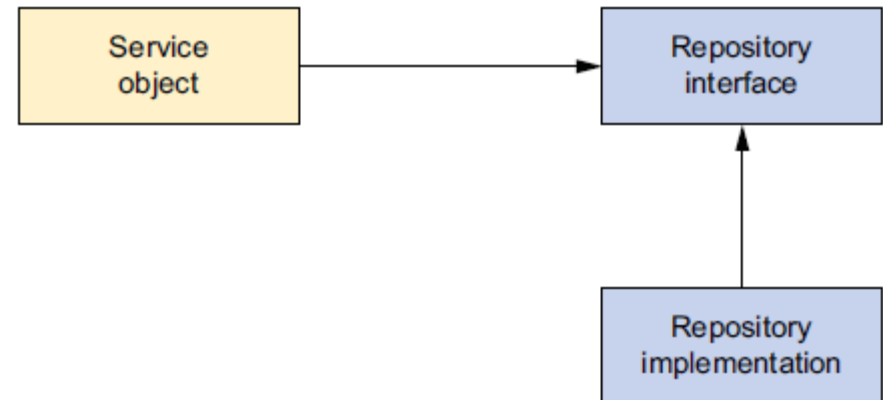
Data Access Object (DAO) pattern

- Data access strategies often differ depending on the storage mechanism.
 - Accessing relational tables is different from accessing XML files
 - JDBC and Hibernate are different ways to access relational tables
- To avoid scattering persistence logic across the application, we factor database access into one or more components.
- Such component is called **Data Access Object (DAO)** or a **repository**.
- DAO is a design pattern that provides abstract interface to the retrieval of data from a data resource.
- It aims to provide a uniform access interface for persistent storages.



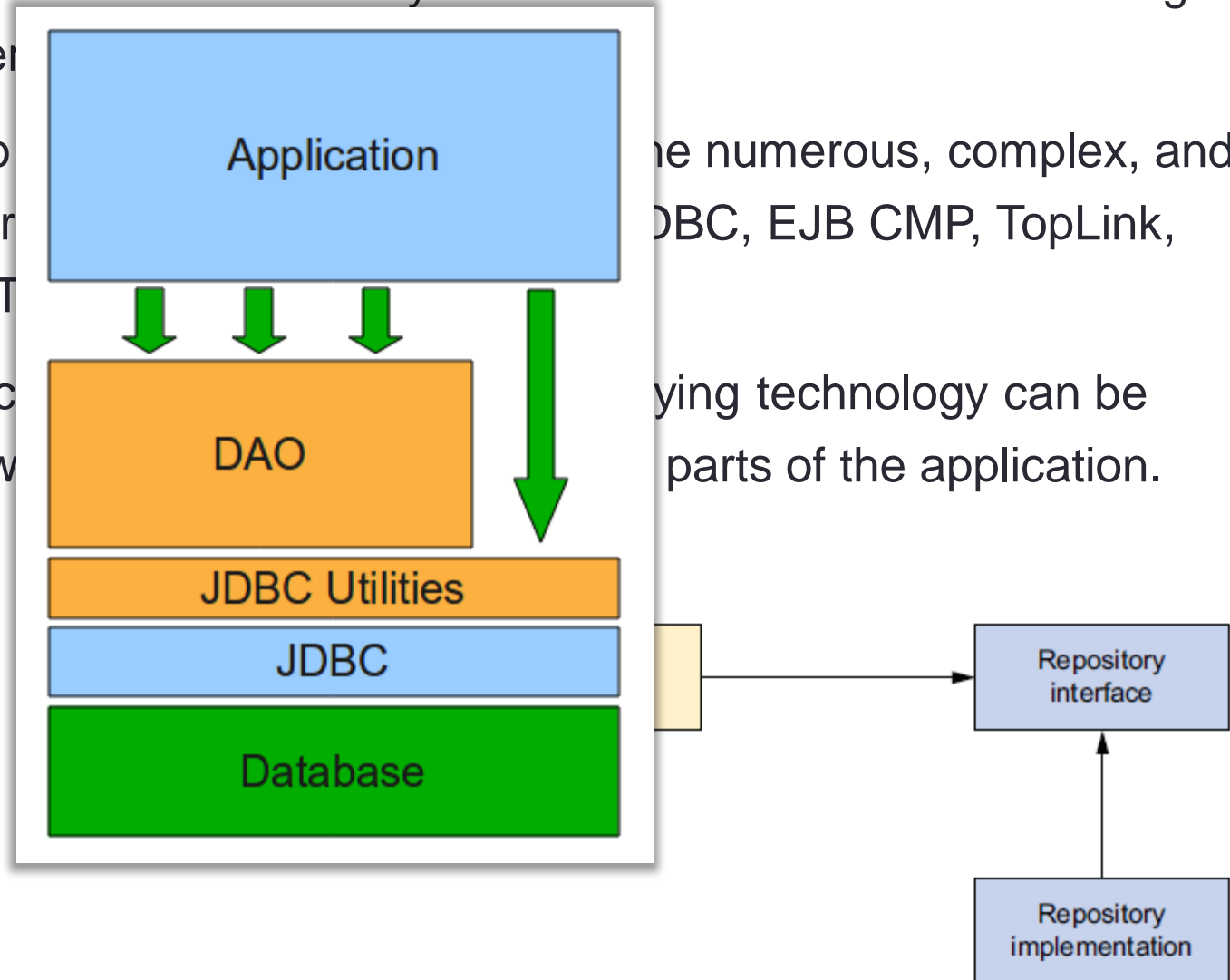
DAO: Advantages

- Changes to persistence access layer do not affect DAO clients as long as the interface remains correctly implemented.
- DAO is used to insulate an application from the numerous, complex, and varied Java persistence technologies (e.g., JDBC, EJB CMP, TopLink, Hibernate, iBATIS).
- Using Data Access Objects means the underlying technology can be upgraded or swapped without changing other parts of the application.



DAO: Advantages

- Changes to persistence access layer do not affect DAO clients as long as the interface remains the same.
- DAO is used to wrap numerous, complex, and varied Java persistence technologies such as JDBC, EJB CMP, TopLink, Hibernate, iBATIS, etc.
- Using Data Access Objects can be upgraded or swapped out without changing technology can be changed without changing parts of the application.



HelloSpringHibernate: Repository interface

GuestBookController.java

Package hkmu.comps380f.dao

GuestBookEntryRepository.java

uses

```
public interface GuestBookEntryRepository {  
  
    public void addEntry(GuestBookEntry e);  
  
    public void updateEntry(GuestBookEntry e);  
  
    public List<GuestBookEntry> listEntries();  
  
    public GuestBookEntry getEntryById(Integer id);  
  
    public void removeEntryById(Integer id);  
}
```

implements

GuestBookEntryRepositoryImpl.java

HelloSpringHibernate: Repository implementation

GuestBookEntryRepositoryImpl.java

```
package hkmu.comps380f.dao;

import java.util.List;
import org.hibernate.Hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.query.Query;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import hkmu.comps380f.model.GuestBookEntry;

@Repository
public class GuestBookEntryRepositoryImpl implements GuestBookEntryRepository {
    @Autowired
    private SessionFactory sessionFactory;

    // ... Implementation of repository methods
}
```

- **@Repository** tells Spring to create a Spring bean for this class.
- **@Autowired** automatically finds the matched Spring bean (by type) and set it as sessionFactory.

Repository implementation: CRUD

- **Create** a new guest book entry:

```
@Override
@Transactional
public void addEntry(GuestBookEntry e) {
    Session session = this.sessionFactory.getCurrentSession();
    session.persist(e);
}
```

GuestBookEntryRepositoryImpl.java

- **Read** all guest book entries:

```
@Override
@Transactional
public List<GuestBookEntry> listEntries() {
    Session session = this.sessionFactory.getCurrentSession();

    String hql = "FROM GuestBookEntry";

    Query query = session.createQuery(hql);
    List<GuestBookEntry> entriesList = query.list();
    return entriesList;
}
```

GuestBookEntryRepositoryImpl.java

Hibernate Query Language

Repository implementation: CRUD (cont')

- **Read** a guest book entry with a particular ID:

```
@Override
```

```
@Transactional
```

```
public GuestBookEntry getEntryById(Integer id) {
```

```
    Session session = this.sessionFactory.getCurrentSession();
```

```
    GuestBookEntry e = (GuestBookEntry) session.load(GuestBookEntry.class, id);
```

```
    Hibernate.initialize(e);
```

```
    return e;
```

```
}
```

```
GuestBookEntryRepositoryImpl.java
```

- Due to lazy loading, the GuestBookEntry object **e** may not be accessible after the Hibernate Session is closed.
- We need to use **Hibernate.initialize(e)** to make sure the object **e** is loaded before the session is closed.

Repository implementation: CRUD (cont')

- **Update** a guest book entry with a particular ID:

```
@Override
@Transactional
public void updateEntry(GuestBookEntry e) {
    Session session = this.sessionFactory.getCurrentSession();
    session.update(e);
}
```

GuestBookEntryRepositoryImpl.java

- **Delete** a guest book entry with a particular ID:

```
@Override
@Transactional
public void removeEntryById(Integer id) {
    Session session = this.sessionFactory.getCurrentSession();
    GuestBookEntry e = (GuestBookEntry) session.load(GuestBookEntry.class, id);
    if (e != null) {
        session.delete(e);
    }
}
```

GuestBookEntryRepositoryImpl.java

HelloSpringHibernate: Controller

GuestBookController.java

```
@Controller
@RequestMapping("guestbook")
public class GuestBookController {
```

@Resource

Work the same as **@Autowired**

```
private GuestBookEntryRepository gbEntryRepo;
```

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

```
@GetMapping("/add")
public ModelAndView addCommentForm() {
    return new ModelAndView("AddComment", "command", new GuestBookEntry());
}
```

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

HelloSpringHibernate: Controller (cont')

GuestBookController.java

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

```
}
```

JDBC vs. ORM

- Relational databases are organized in rows and columns.
- Using JDBC:
 - It is tedious to read data from database in a row-by-row manner to OO programs.
 - It is also tedious to update data from OO programs to database.
- Using ORM tool (e.g., Hibernate):
 - ORM tool maps data in relational database to entity object.
 - It simplifies the data reading and updating operations in OO programs.

Java Persistence API (JPA) vs. Hibernate

- **Hibernate** is an **ORM tool**.
 - Other ORM tool examples: TopLink, Java Data Objects (JDO).
- **Java Persistence API (JPA)** is a specification of an **interface**. You can choose any implementation available on the market. E.g.,
 - The programs can be written according to the JPA interface.
 - We can configure our programs to use **Hibernate** as the implementation, and switch to another implementation (e.g., EclipseLink) later.
- If a program is written to use **Hibernate** without JPA, it is locked in and switching to use another persistence provider requires **a fair amount of code change**.

JPA Components

- **ORM**: mechanism to map objects to relational data
- **Entity manager (~ Hibernate Session)** to perform CRUD operations (Create, Read, Update and Delete)
- **Java Persistence Query Language (JPQL)** to retrieve data with an OO query language
- Transaction and locking mechanism to protect data from being corrupted under concurrent access
- Callbacks and Listeners
- When using **JPA**, we still need **a lot of boilerplate code** for interacting directly with the Entity Manager.
- **Spring Data JPA** is a better solution.

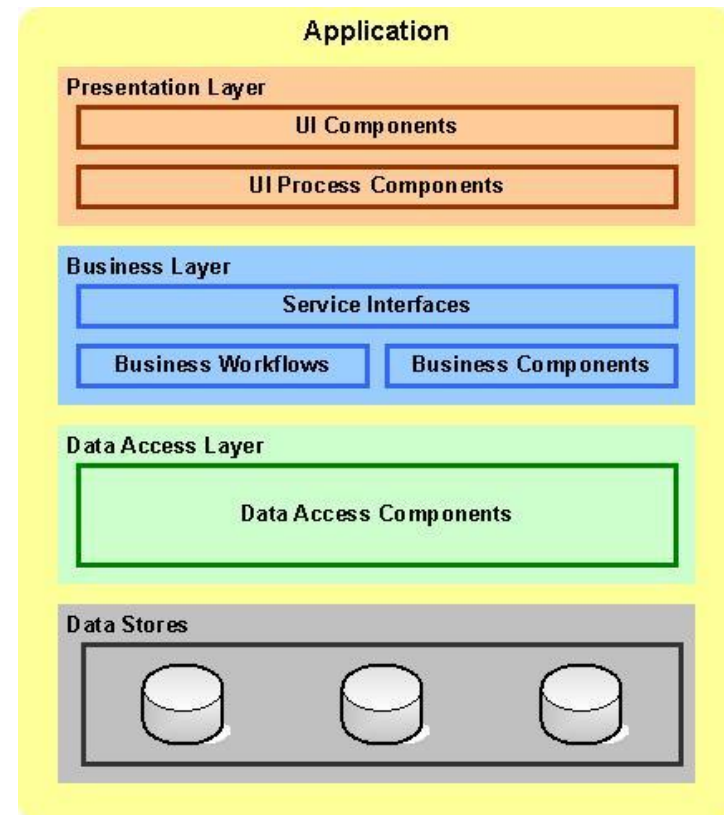
Spring Data

Motivations:

- Best practice indicates you should have Data Access Components or a Data Service Layer.
- People often make simple mistakes interacting with EntityManager.
- Different EntityManager API for different stores.

Spring Data:

- An open-source project managed by SpringSource.
- Rely on Spring Framework.
- Provide a set of common patterns for persisting data using existing libraries.
- **Provide a JPA Repository Interface**
- **Automatic JPA Repository Implementation**



Spring Data JPA with Hibernate

Webapp example: *HelloSpringDataJPA*

Maven dependencies in **pom.xml**

- In addition to the dependencies for Derby DB, spring-jdbc, spring-orm, spring-tx, hibernate-core, hibernate-entitymanager

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-jpa</artifactId>
  <version>2.6.2</version>
  <exclusions>
    <exclusion>
      <artifactId>spring-asm</artifactId>
      <groupId>org.springframework</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

Spring Data JPA with Hibernate

dispatcher-servlet.xml

```
<beans ...
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="...
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
">
...
<!-- Create default configuration for Hibernate -->
<bean id="hibernateJpaVendorAdapter"
  class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>

<!-- Configure the entity manager factory bean -->
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter" ref="hibernateJpaVendorAdapter"/>
  <!-- Set JPA properties -->
  <property name="jpaProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
  <!-- Set base package of your entities -->
  <property name="packagesToScan" value="hkmu.comps380f.model"/>
</bean>
```


Spring Data JPA with Hibernate (cont')

dispatcher-servlet.xml

```
<!-- Configure the transaction manager bean -->
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<!-- Enable annotation driven transaction management -->
<tx:annotation-driven/>

<!-- Configure Spring Data JPA and set the base package of the
      repository interfaces -->
<jpa:repositories base-package="hkmu.comps380f.dao"/>
```

Spring Data JPA: Automatic JPA Repository

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
GuestBookEntryRepository.java
```

```
public interface GuestBookEntryRepository extends JpaRepository<GuestBookEntry, Integer>{  
}
```

- JpaRepository is parameterized such that it knows
 - This is a repository for persisting **GuestBookEntry** objects, and
 - **GuestBookEntry** objects have an ID of type **Integer**.
- We only need to define the above interface and we **do not need to implement repository methods**.
- Spring Data JPA will **automatically generate** a set of CRUD implementations, including
 - findAll, count, delete, deleteById, deleteAll, exists, findById, save
 - See this link for the complete list:
<http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>

Spring Data JPA: Automatic JPA Repository (cont')

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
GuestBookEntryRepository.java
```

```
public interface GuestBookEntryRepository extends JpaRepository<GuestBookEntry, Integer>{  
}
```

Spring Data JPA will *automatically generate* CRUD implementations, e.g.,

- GuestBookEntry **save**(GuestBookEntry entity)
 - Insert a new entry into the table
- GuestBookEntry **findById**(Integer id)
 - Find the entry with the given id
- List<GuestBookEntry> **findAll**()
 - Find all the guestbook entries in the table
- void **delete**(GuestBookEntry entity)
 - Delete the guestbook entry from the table
- Besides generated methods, we can also implement our own methods.

- E.g.,

- This method will give all guestbook entries from the specified user.
- Naming convention: **Query verb** **Predicate**



More details: <http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>

Define repository method 2: By @Query

- We can also create a method in the repository interface by giving custom **JPQL query**. Spring Data JPA will automatically generate its implementation.
- E.g.,

```
@Query("select s from Spitter s where s.email like '%gmail.com' ")  
List<Spitter> findAllGmailSpitters();
```

- This method is also useful when the naming convention gives a long method name and you want to replace it with a shorter one.

Define repository method 3: By implementation

- We can follow the **traditional Hibernate way** to define a repository method.

```
public class SpitterRepositoryImpl implements SpitterSweeper {
    @PersistenceContext
    private EntityManager em;

    public int eliteSweep() {
        String update =
            "UPDATE Spitter spitter " +
            "SET spitter.status = 'Elite' " +
            "WHERE spitter.status = 'Newbie' " +
            "AND spitter.id IN (" +
            "SELECT s FROM Spitter s WHERE (" +
            "SELECT COUNT(spittles) FROM s.spittles spittles) > 10000" +
            ")";
        return em.createQuery(update).executeUpdate();
    }
}
```

```
public interface SpitterSweeper {
    int eliteSweep();
}
```

```
public interface SpitterRepository
    extends JpaRepository<Spitter, Long>, SpitterSweeper {
    ....
}
```

HelloSpringDataJPA: Controller

GuestBookController.java

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

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

    @GetMapping("/add")
    public ModelAndView addCommentForm() {
        return new ModelAndView("AddComment", "command", new GuestBookEntry());
    }

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

Generated by configuration
<jpa:repositories ... />

HelloSpringDataJPA: Controller (con't)

GuestBookController.java

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

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

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