Hibernate

Hibernate Framework

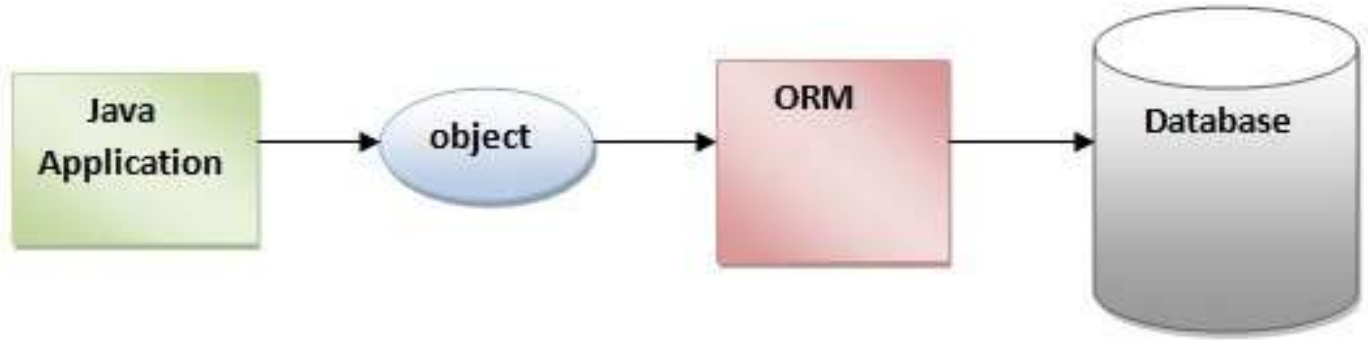
- Hibernate is a Java framework that simplifies the development of Java application to interact with the database.
- It is an open source, lightweight, ORM (Object Relational Mapping) tool.
- Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

- Hibernate bu Framework bo’lib u java da database bilan ishlashni osonlashtiradi.
- Bu ochiq manbali, engil, ORM (Object Relational Mapping) qo’llaydigan vositasi (asbob,uskuna).
- Hibernate malumotlar bilan ishlash uchun JPA (Java Persistence API) spetsifikatsiyalar larini realizatsiya qilgan.

- dasturlash.uz

ORM Tool

- ORM - stands for **O**bject-**R**elational **M**apping (ORM) – Ob'ekt ni bog'liqlarini to'g'irlash.
- ORM is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.
- ORM - bu bazadagi malumotlarni OOP dagi ob'ekt ko'rinishiga o'giradigan dasturlash texnikasi. U java, C# va shunga o'xshash tillarda mavjut.



Java ORM Frameworks

- There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.
  - Enterprise JavaBeans Entity Beans
  - Java Data Objects
  - Castor
  - TopLink
  - Spring DAO
  - Hibernate
  - And many more
- Java da ORM ni realizatsiya qilgan birnechta framework lar mavjut. ORM lar bular ob’ektlarni bazaga saqlaydi va bazadan malumotlarini ob’ekt ko’rinishida oladi.

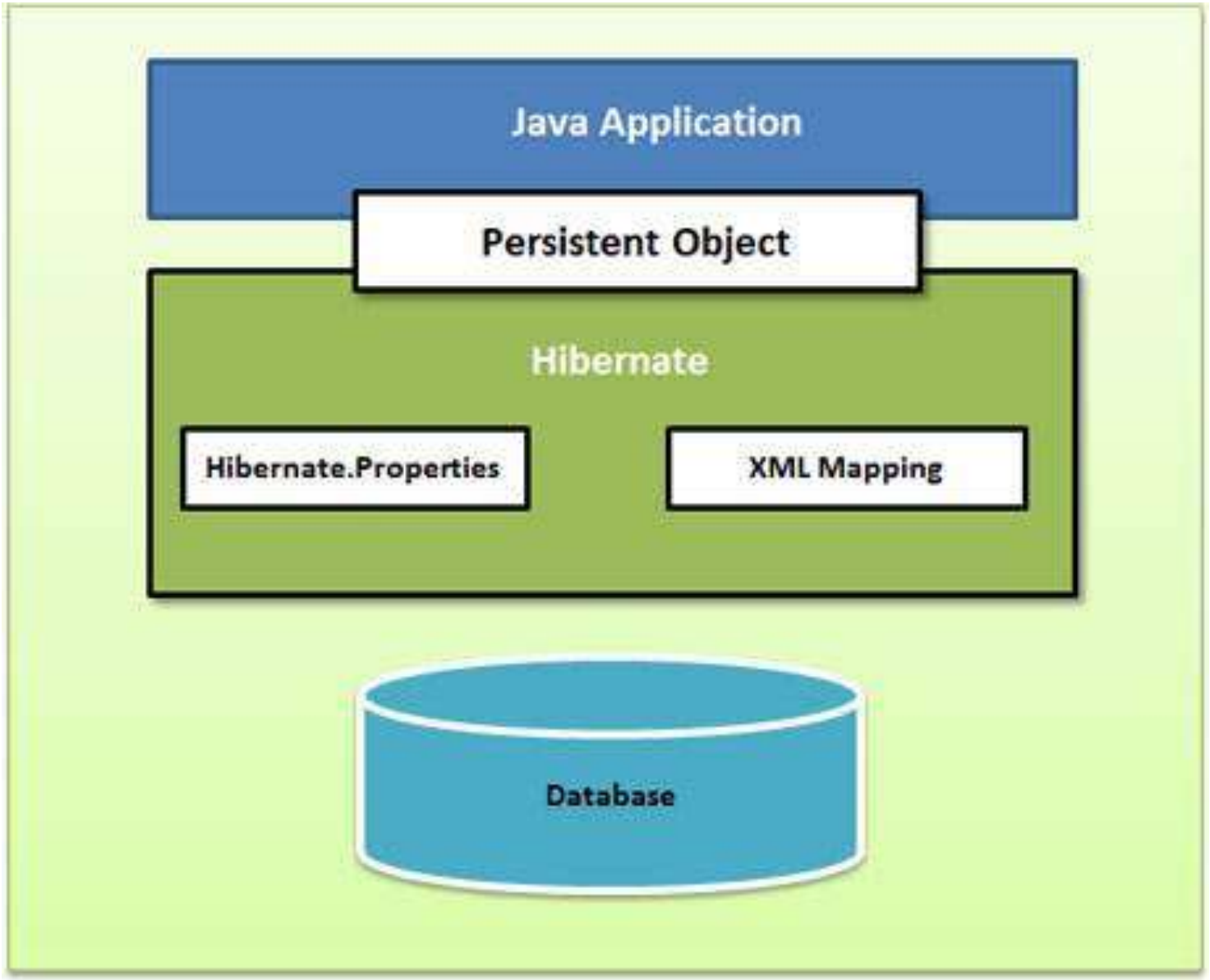
- dasturlash.uz

JPA

- Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools.
- JPA is a set of specifications for accessing, persisting, and managing data between Java objects and relational database entities.
- The **javax.persistence** package contains the JPA classes and interfaces
- Hibernate does support the Java Persistence API **(JPA)** specification.
- JPA – bu javada ORM ni qanday realizatsiya qilish kerak ekanligi haqida qoidalar to'plami. Yani spesifikatsiya lar to'plami.
- **javax.persistence** package da JPA class va inerface lari mavjut.
- Hibernte JPA spesificatsiyalarni implementatsiya qilgan.

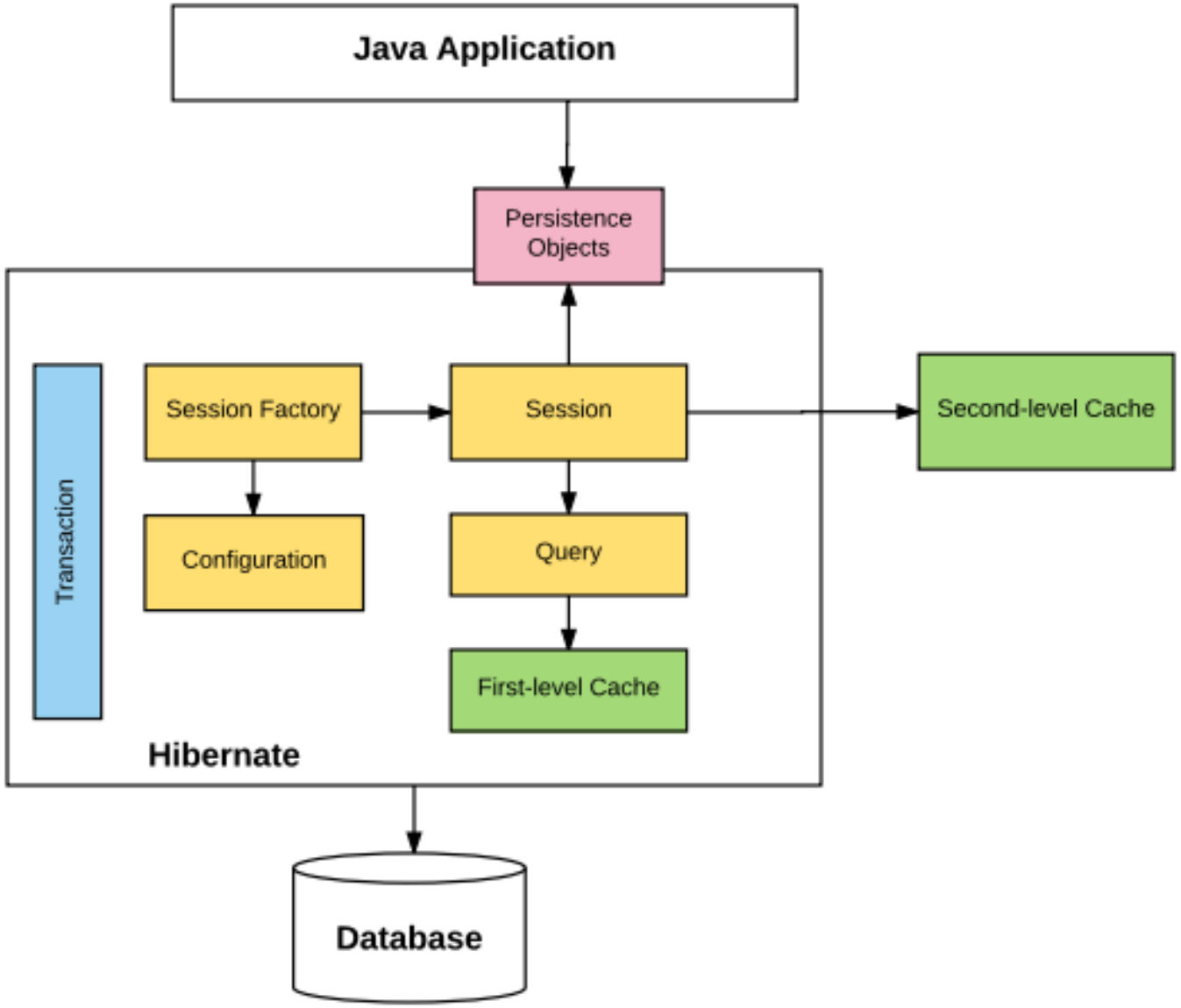
- dasturlash.uz

Hibernate Architecture - simple



- dasturlash.uz

Hibernate Architecture – more detail



dasturlash.uz

Advantages of Hibernate Framework

- 1) Open Source and Lightweight
- 2) Fast Performance
- 3) Database Independent Query
- 4) Automatic Table Creation
- 5) Simplifies Complex Join
- 6) Provides Query Statistics and Database Status
  - Hibernate supports Query cache and provide statistics about query and database status.

Hibernate Architecture

- Configuration.
  Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class annotated with @Configuration. It is used by Session Factory to work with Java Application and the Database. It represents an entire set of mappings of an application Java Types to an SQL database

- Session Factory
  Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.

- Session
  This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class. The instance of a session can be retrieved from the SessionFactory bean.

- Query
  It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query database, including NamedQuery and Criteria API.

Hibernate Architecture

- First-level cache
  It represents the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.

- Transaction
  It is enables you to achieve data consistency, and rollback incase something goes unexpected.

- Persistent objects
  These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate.They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.
- Second-level cache : It is used to store objects across sessions.

Libs

```
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
      <version>5.4.30.Final</version>
</dependency>
```

```
 <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.2.19</version>
 </dependency>
```

Configuration – for PostgreSQL XML

In hibernate.cfg.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-configuration>
    <session-factory>
   <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL82Dialect</property>
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.username">hibernate_lesson_user</property>
        <property name="hibernate.connection.password">hibernate_lesson_password</property>
        <property name="hibernate.connection.url">
            jdbc:postgresql://localhost:5432/hibernate_db_lesson
        </property>
       <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">update</property>
        <property name="show_sql">true</property>



        <!-- MAPPINGS -->
        <mapping class =“….”  or  resource=“…”   />
</session-factory>


</hibernate-configuration>
```

- dasturlash.uz

Configuration – for PostgreSQL Annotation

POJO - Domain

```
public class Employee {

    private Integer id;
    private String firstName;
    private String lastName;

}
```

- dasturlash.uz

Mapping – using xml

In employee.hbm.xml file:

```
<?xml version='1.0' encoding='UTF-8'?>

<hibernate-mapping>
    <class name="com.company.Employee" table="employeeJon">
        <id name="id">
            <generator class="assigned"></generator>
        </id>

        <property name="firstName"></property>
        <property name="lastName"></property>

    </class>

</hibernate-mapping>
```

In hibernate.cfg.xml add

```
<mapping resource="employee.hbm.xml"/>
```

Annotation Based

```
@Entity
@Table(name = “employeejon")
public class Employee {

    @Id
    private Integer id;

    private String firstName;

    private String lastName;

}
```

- dasturlash.uz

Mian

```
StandardServiceRegistry ssr = new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

    Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();

    SessionFactory factory = meta.getSessionFactoryBuilder().build();
    Session session = factory.openSession();
    Transaction t = session.beginTransaction();

    Employee e1=new Employee();
    e1.setId(105);
    e1.setFirstName("Gaurav");
    e1.setLastName("Chawla");

    session.save(e1);
    t.commit();

    System.out.println("successfully saved");

    factory.close();
    session.close();
```

- dasturlash.uz

Hibernate Annotations

- Hibernate Annotations are based on the JPA 2 specification and supports all the features.
- All the JPA annotations are defined in the **javax.persistence** package. Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification.
- The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

Create the Persistence class.

- **@Entity** annotation marks this class as an entity.
- **@Table** annotation specifies the table name. If you don't use @Table annotation, hibernate will use the class name as the table name by default.
- **@Id** annotation marks the identifier for this entity. (Primary Key)
- **@Column** annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name by default.

@GeneratedValue

- If we want the primary key value to be generated automatically for us, **we can add the *@GeneratedValue* annotation**.
- This can use 4 generation strategy: AUTO, IDENTITY, SEQUENCE, TABLE.
- If we don't specify a value explicitly, the generation type defaults to AUTO.

GenerationType.IDENTITY

- It relies on an auto-incremented database column and lets the database generate a new value with each insert operation
- From a database point of view, this is very efficient because the auto-increment columns are highly optimized, and it doesn't require any additional statements.
- One thing to note is that IDENTITY generation disables batch updates.
- It Creates Separate sequence for each table (tablename_id_seq)

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
```

- dasturlash.uz

GenerationType.SEQUENCE

- It uses a database sequence to generate unique values.
- It requires additional select statements to get the next value from a database sequence. But this has no performance impact for most applications.
- If you don’t provide any additional information, Hibernate will request the next value from its default sequence.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Long id;
```

- dasturlash.uz

@SequenceGenerator

- A sequence is a database object that can be used as a source of primary key values.

- The *@SequenceGenerator* annotation lets you define the name of the generator, the name, and schema of the database sequence and the allocation size of the sequence.
- @SequenceGenerator has following elements :
  'name', 'catalog', 'schema', 'sequenceName', 'initialValue' and 'allocationSize'.
  All elements of @SequenceGenerator are optional except for 'name' element.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book_generator")
@SequenceGenerator(name="book_generator", sequenceName = "book_seq")
private Long id;
```

GenerationType.AUTO

- The *GenerationType.AUTO* is the default generation type and lets the persistence provider choose the generation strategy.

- If you use Hibernate as your persistence provider, it selects a generation strategy based on the database specific dialect. For most popular databases, it selects *GenerationType.SEQUENCE* which I will explain later.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

GenerationType.TABLE

- The *GenerationType.TABLE* gets only rarely used nowadays.
- It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order.
- This slows down your application

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
private Long id;
```

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "book_generator")
@TableGenerator(name="book_generator", table="id_generator", schema="bookstore")
private Long id;
```

UUID

UUID

```
@Id
@GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid", strategy = "uuid")
@Column(name = "uuid", unique = true)
private String uuid;
```

- We can use uuid as an primary key.

```
@Id
@GeneratedValue(generator = "UUID")
@GenericGenerator(
        name = "UUID",
        strategy = "org.hibernate.id.UUIDGenerator"
)
@Column(name = "id", updatable = false, nullable = false)
private String id;
```

UUID using UUID class

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "id", nullable = false)
protected UUID id;
```

UUID using Hibernate and UUID class

```
@Id
 @GeneratedValue(generator = "UUID")
 @GenericGenerator( name = "UUID", strategy = "org.hibernate.id.UUIDGenerator")
 @Column(name = "id", updatable = false, nullable = false)
 private UUID id;
```

@Column

- **name** : permits the name of the column to be explicitly specified—by default, this would be the name of the property.
- **length** : permits the size of the column used to map a value (particularly a String value) to be explicitly defined. The column size defaults to 255.
- **nullable** : permits the column to be marked NOT NULL when the schema is generated.
- **unique** : permits the column to be marked as containing only unique values.
- **columnDefinition**: The SQL fragment that is used when generating the DDL for the column.

```
@Column(name="DESC", columnDefinition="CLOB NOT NULL", table="EMP_DETAIL")
@Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
@Column(name = "description", columnDefinition = "varchar(50) default ‘some qiymat‘ ")
```

- dasturlash.uz

@Column - Example

```
@Entity
@Table(name = "ish_test")
public class IshTest {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name", nullable = false, length = 100, unique = true)
    private String name;

    @Column(name = "description", columnDefinition = "varchar(50) default 'some qiymat'")
    private String description;

    Def-Constructor…
    Gettet-Setter…
}
```

- dasturlash.uz

@Column - In SQL

```
CREATE TABLE ish_test
(
    id bigint NOT NULL DEFAULT nextval('ish_test_id_seq'),
    name character varying(100)    NOT NULL UNIQUE,
    description character varying(50)  DEFAULT 'some    qiymat‘ ,


    CONSTRAINT ish_test_pkey PRIMARY KEY (id),
    CONSTRAINT uk_mqf7qrckdek8x46ny3imedno
)
```

- dasturlash.uz

- dasturlash.uz