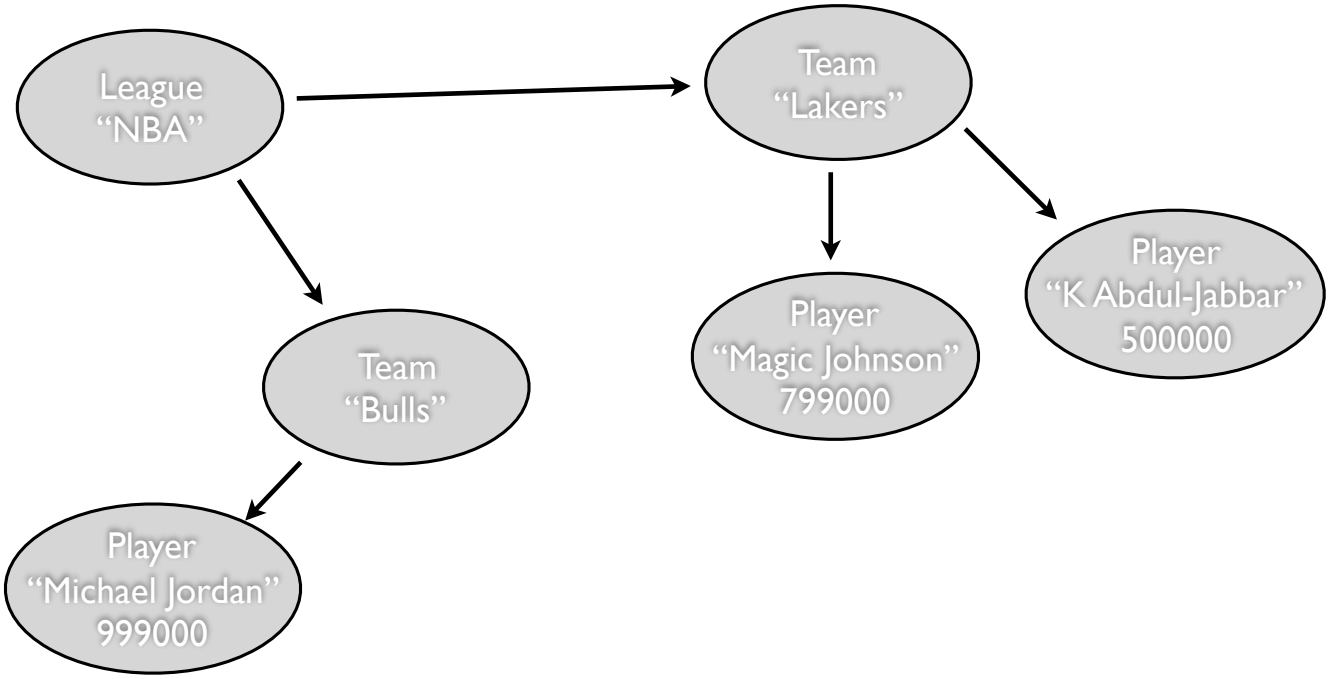
OO ja persistence

- Application state is captured by the object graph:



OO ja persistence

- Serialization (we have done this)
- Object-relational mapping
  - map classes to database tables
    - map instance variables to table columns (to identify an instance we also need an `id`)
    - references to other objects contain the table name and instance identification (`id`)
- see http://en.wikipedia.org/wiki/Object-relational_mapping

ORM challenges

- Inheritance
- Equality - `a == b` ja `a.equals(b)` vs. `id` in relational model
- Navigation - follow references in Java vs. select in SQL
- see
  - http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
  - http://www.hibernate.org/about/orm

Hibernate

- Either annotate classes and generate tables from that or use existing tables as a baseline
- Use either annotations (or define class persistence in an XML config file)
- For tutorials and documentation, see http://hibernate.org/orm/
- (Note: Netbeans+Glassfish bundle has hibernate 4.3.1)

Annotated class

```
@Entity
public class User {
    private Long id;
    private String name;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Class after `@Entity` is to be persisted

Field whose getter follows `@Id` (`id`) is going to be primary key ie. unique identifier

This field will have value generated by the database (`@GeneratedValue`)

For more on JPA annotations, see https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html#entity-mapping section 2.2

To-many relationship

```
@Entity
public class Team {
    private Long id;
    private String name;
    private List<Person> members;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    …

    @OneToMany(targetEntity=Person.class,
               fetch=FetchType.EAGER,
               cascade=CascadeType.ALL)
    public List<Person> getMembers() {
        return members;
    }
    public void setMembers(List<Person> members) {
        this.members = members;
    }

}
```

Class after `@Entity` is to be persisted

Field whose getter follows `@Id` (`id`) is going to be primary key ie. unique identifier

To-many mapping is described by declaring the `targetEntity` class. Additional parameters declare fetch strategy (`EAGER` or `LAZY`) and cascade rule

Simple set up

Create Configuration that can be used for defining properties of mapping. Here two persistent classes are specified in config file (more later) and one here as an example

Create schema for the classes to be persisted. Creation script is both run and printed to log.

```
Configuration config = new Configuration();
config.addAnnotatedClass(mypackage.Person.class);
config.addAnnotatedClass(mypackage.Team.class);
config.configure();

new SchemaExport(config).create(true, true);

StandardServiceRegistryBuilder serviceRegistryBuilder =
  new StandardServiceRegistryBuilder();
serviceRegistryBuilder.applySettings(config.getProperties());
ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
this.sessionFactory = config.buildSessionFactory(serviceRegistry);
```

Consider placing this code into a singleton and create a getter for session factory.

Create factory that will provide Session objects inside which persistence operations take place

Saving objects

… and open a session.

```
Session session =
  HibernateStuff.getInstance().getSessionFactory().openSession();

session.beginTransaction();

User u = new User();
u.setName("Pekka");

Team t = new Team();
t.setName("superteam");

… more operations on application objects …

session.saveOrUpdate(t); // save changes in object graph starting at t
session.saveOrUpdate(u): // save changes in object graph starting at u

session.getTransaction().commit();
```

Queries

- Three ways:
  - SQL
  - HQL (Hibernate Query Language)
  - Criteria API

Criteria API example

```
Team steam = new Team();
steam.setName("Ducks");
Example teamLikeThis = Example.create(steam);

Criteria crit = session.createCriteria(Team.class);
crit.add(teamLikeThis);

List res = crit.list();
```

```
List persons = session.createCriteria(Person.class)
            .add(Restrictions.le("age", maxage))
            .list();
```

- For more on criteria API, see https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/querycriteria.html
- Note: New criteria API (with perhaps a bit steeper learning curve) is available and this one is getting deprecated. Check latest Hibernate documentation if you are planning to do serious hibernate development.

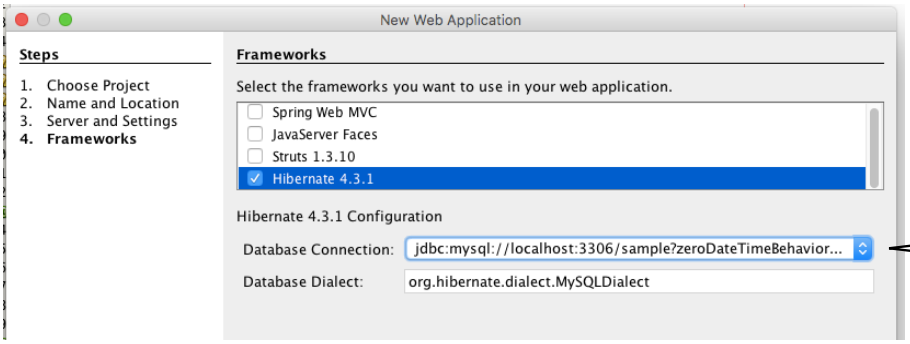
Misc: app startup & shutdown

@WebListener to tell the framework this class implements the lifecycle methods.

```
@WebListener
public class StartupListener implements javax.servlet.ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("StartupListener contextInitialized()");

        HibernateStuff.getInstance();
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("StartupListener contextDestroyed()");
    }
}
```

HibernateStuff singleton exports schema in its constructor and makes session factory available

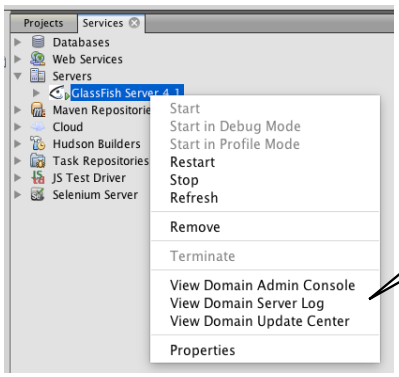
Misc: create project in Netbeans



Create a web application project and proceed until frameworks selection - specify mysql for database connection.

Check your hibernate.cfg.xml for settings such as db username and password. (Note: don't use this example directly)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate
<hibernate-configuration>
  <session-factory>
    <property name="show_sql">true</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/sample</property>
    <property name="connection.username">myuser</property>
    <property name="connection.password">qwerty</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <property name="hibernate.query.factory_class">org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory</property>
  </session-factory>
</hibernate-configuration>
```



Reading server log will be helpful.

Download mysql connector and add into your project.