Helianto core module

A word on complexity

If you are reading this document from the beginning, you may likely be asking yourself if Helianto pays off. Considering the world of the ever-changing requirements, your experience with Helianto will show you, somehow, that YES, it has worthed. But if you are unfamiliar with concepts like dependency injection and object relational mapping, you may not give even it a try. Since this project was published as open source (2006), such technologies have increasingly grown to play a major role.

Helianto can be taken as a shortcut in the path for learning and adopting this way of programming. It is a “learn as you go” alternative that encourages use of agile techniques and sound programming practices, like test driven development (TDD), clear interface definitions and separation of concerns.

Therefore, an important guideline for writing the reference, specially this core module, is to go beyond the plain description of key features and also say something valuable for beginners. Code samples and short explanations are kept as a minimum to facilitate reading on, but are surely not enough to cover all what is need to a full understanding of the presented subject.

For some odd words to beginners, here is a list o helpful links:

- Dependency injection (DI): http://martinfowler.com/articles/injection.html , http://en.wikipedia.org/wiki/Dependency_injection .
- Object relational mapping (ORM): http://en.wikipedia.org/wiki/Object-relational_mapping .
- Java persistence API (JPA): http://en.wikipedia.org/wiki/Java_Persistence_API .

Infrastructure

Infrastructure in helianto-core module provides for Spring based dependency injection and JPA object relational mapping.

Helianto relies on Hibernate (alternatively JPA) to keep information on a SQL database. A full discussion on object relational mapping advantages and penalties is beyond the scope of this documentation. The first few things to note on a new project extending Helianto are:

1. **Helianto focuses on Java classes**: database tables are not as important as domain classes they are mapped to and they will seldom be discussed. The helianto-core module configuration handles the automatic creation of tables, primary keys, unique constraints and foreign keys required by the persistent domain classes through JPA compatible annotations.

2. **Domain classes have JPA annotations and are registered: y**ou can easily create new domain classes, but you should first be aware (check the section below) how do they relate to the isolation classes provided by Helianto.
3. **Data access objects, also called repositories**: (although many developers may prefer to use Hibernate or JPA artifacts directly) are provided for specific domain hierarchies. They inherit from *AbstractBasicDao<T>* or *AbstractFilterDao<T, F extends Filter>* to expose a convenient subset of persistence operations. Wherever a filter is implemented, a kind of specialized value object, its corresponding *FilterDao* is capable to create queries without any SQL handwriting (nor HQL or JPQL).
4. **[NEW] Spring 3 repository factories**: are used to easily create DAOs and register them to the Spring root application context. DAOs are excellent candidates for dependency injection, a key feature of the Spring framework.

The following code fragments illustrate some of the above features.

CODE EXAMPLE: Category domain class

```
/**
 * Categories.
 *
 * @author Mauricio Fernandes de Castro
 */
@javax.persistence.Entity
@Table(name="core_category",
    uniqueConstraints = {@UniqueConstraint(columnNames={"entityId", "categoryGroup", "cat
)
public class Category implements java.io.Serializable {

    private static final long serialVersionUID = 1L;
    private int id;
    private Entity entity;
    private int categoryGroup;

    // ... code ommited for brevity ...

    /**
     * Category entity.
     */
    @ManyToOne
    @JoinColumn(name="entityId")
    public Entity getEntity() {
        return this.entity;
    }
    public void setEntity(Entity entity) {
        this.entity = entity;
    }

    /**
     * Group of categories.
     */
    public int getCategoryGroup() {
        return categoryGroup;
    }
    public void setCategoryGroup(int categoryGroup) {
        this.categoryGroup = categoryGroup;
```

```
    }
    public void setCategoryGroup(CategoryGroup categoryGroup) {
        this.categoryGroup = categoryGroup.getValue();
    }

    /**
     * Category code.
     */
    @Column(length=20)
    public String getCategoryCode() {
        return this.categoryCode;
    }
    public void setCategoryCode(String categoryCode) {
        this.categoryCode = categoryCode;
    }

    // ... code ommited for brevity ...

    /**
     * equals
     */
    @Override
    public boolean equals(Object other) {
        if ( (this == other ) ) return true;
        if ( (other == null ) ) return false;
        if ( !(other instanceof Category) ) return false;
         Category castOther = (Category) other;

        return ((this.getEntity()==castOther.getEntity())
                        || ( this.getEntity()!=null && castOther.getEntity()!=null
                                        && this.getEntity().equals(castOther.getEntity()))
            && (this.getCategoryGroup()==castOther.getCategoryGroup())
            && ((this.getCategoryCode()==castOther.getCategoryCode())
                        || ( this.getCategoryCode()!=null && castOther.getCategoryCode()!
                                        &&
this.getCategoryCode().equals(castOther.getCategoryCode()) ));
    }

    /**
     * hashCode
     */
    @Override
    public int hashCode() {
        int result = 17;
        result = 37 * result + (int) this.getCategoryGroup();
        result = 37 * result + ( getCategoryCode() == null ? 0 : this.getCategoryCode().h
        return result;
    }

}
```

The highlighted code shows that:

1. Standard JPA annotations like *@javax.persistence.Entity* and *@javax.persistence.Table* are often required, as well as *@javax.persistence.ManyToOne* and others.
2. The *@javax.persistence.Table* annotation includes at least one unique constraint

definition; this is relevant for the DAO definition.

3. Any persistent domain class is a plain Java bean.
4. The methods equals and hashCode are required; DO NOT FORGET THEM!

Some domain classes are part of a hierarchy and have additional mapping.

CODE EXAMPLE: UserGroup domain class

```
/**
 *
 * An user account (or group) represents a set of roles within an <code>Entity</code>.
 *
 * @author Mauricio Fernandes de Castro
 */
@javax.persistence.Entity
@Table(name="core_user",
    uniqueConstraints = {@UniqueConstraint(columnNames={"entityId", "userKey"})}
)
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="type",
    discriminatorType=DiscriminatorType.CHAR
)
@DiscriminatorValue("G")
public class UserGroup implements java.io.Serializable, Comparable<UserGroup>, NaturalKeyI
{
    // ... code ommited for brevity ...
```

The highlighted code shows that:

1. All members of the hierarchy are stored to the same table, *core_user*, having a column named *type* to distinguish classes.
2. UserGroup has the char value of 'G'.

UserGroup is the root of a hierarchy and may have a DAO like *UserGroupDao*.

CODE EXAMPLE: User domain class

```
/**
 * <p>
 * The user account.
 * </p>
 * <p>
 * An user account represents the association between an <code>Identity</code>
 * and an <code>Entity</code>. Such association allows for a singly identified
 * actor, i.e. a person or any other organizational <code>Identity</code>, to
keep
 * a single authentication scheme and have multiple authorization schemes, one
per
 * <code>Entity</code>.
 * </p>
 * @author Mauricio Fernandes de Castro
 *
 */
@javax.persistence.Entity
@DiscriminatorValue("U")
public class User extends UserGroup implements java.io.Serializable {

    // ... code ommited for brevity ...
```

Other members of the hierarchy are easier to annotate and to maintain. The *User* class is persisted with the *UserGroupDao*.

Helianto does not require much more than this few annotations briefly shown here to annotate domain classes, although there are many details of the JPA specification not covered in detail in this document. You may find more examples by browsing the code for each module root package, where domain classes are usually placed.

Alternative ORM engines

The default infrastructure configuration relies on Hibernate to persist data. It is possible to plug in any other JPA engine.

Installation types and isolation levels

After deploying any Helianto based war file to a web container like Tomcat, the infrastructure beans configured in Helianto create database tables according to classes defined as shown above. However, before an user is authorized into the system, certain instances must be created and persisted. This is going to be referred as installation.

Domain classes are designed so that Helianto may be installed and used in any of the following use cases:

1. Accounts, customers, equipment, etc. belong all to the same company, organization or business entity; this is the simplest use case, the so called stand-alone (or single entity) installation.
2. Users may choose to login to organization A or B, and data from one entity must not

be visible to users logged to any other organization; this is the single namespace installation.

3. Groups of organizations share common instances, and there is more than one group of organizations that can be managed separately; this is the multi namespace installation.

Whenever is necessary to keep different organizations or namespaces under the same database, care must be taken to avoid data collision. The core module has domain classes responsible for laying out the main isolation strategies:

1. **Entity isolation**, meaning that accounts, customers, equipment, etc. from one business entity is not accessible to some other business entity. This is the main responsibility assigned to the class org.helianto.core.Entity.
2. **Namespace isolation**, meaning that common instances to a group of entities (like provinces, parameters, etc.) can be managed as a unit. This is the main responsibility assigned to the class org.helianto.core.Operator. Any Entity holds a reference to its Operator.

ATTENTION!

Do not use javax.persitence.Entity where you may actually require org.helianto.core.Entity. If you are writing code using an IDE like Eclipse or Spring STS, the auto-complete feature may lead you to such mistake. This is not easy to fix because the Entity class is a dependency to almost all classes in this project. If Helianto have been created after the first JPA specification, a better name like BusinessEntity would have been used.

Isolation immediate consequences are:

1. At least one instance of the Entity class and one instance of the Operator class must exist before you can use Helianto no matter what type of installation is required; application and presentation layer must be responsible for hiding domain classes that are not relevant in a particular use case.
2. All Helianto classes, except two or three, have an Operator instance as an aggregate, or have it aggregated in a nth level.
3. Almost all, presumably 90% of the Helianto classes have an Entity as an aggregate, or have it aggregated in a nth level.
4. Having such associations, it is easy to sort out instances by Entity or Operator.

CODE REFERENCE

Check the **org.helianto.core.filter** package for interfaces and classes to help sorting out instances by Entity or Operator as well as any other criterion.

CODE REFERENCE

Check the **org.helianto.core.standalone** package for beans to help create and configure sensible defaults.