

# **Helianto project**

## **Reference Documentation**

**Maurício Fernandes de Castro**

---

# **Helianto project : Reference Documentation**

by Maurício Fernandes de Castro

---

---

---

# Table of Contents

Preface .....	v
1. The Helianto Project .....	1
Introduction .....	1
Mission .....	1
Acknowledgements .....	1
Alternatives .....	2
Software Requirements .....	2
Architecture .....	2
Core functionality .....	2
Overview .....	3
Basic configuration .....	3
The helianto-core package .....	3
The Credential domain object .....	4
The Entity domain object .....	5
The User domain object .....	5
Generic DAO .....	6
Process functionality .....	6
The helianto-process package .....	6
The Process domain model .....	6
Sales functionality .....	7
The helianto-sales package .....	7
The Sales domain model .....	7
Finance and accountability functionality .....	7
A template application including access control .....	7
2. Appendices .....	8
Appendix A - Spring Framework and Dependency injection .....	8
Appendix B - Hibernate and Object relational mapping (ORM) .....	11
Appendix C - Acegi Security .....	13
Security object .....	13

---

# Preface

The Helianto Project was born after a discussion on how to leverage the competitive advantage of a small group of small organizations having different competences in the software business. The key value behind the Helianto mission development was to promote collaboration in disciplines where all members might have common requirements.

The name "Helianto" derives from the latin word for the sunflower plant. The idea showed up from an original sketch, where a couple of functional modules were drawn in a central corolla, surrounded by as many application "petals" as desired. The name still remains associated to the sunflower geometry, although the project focuses only the corolla.

Helianto is not meant to be a product to be used out of the box. Actually, it requires customization and some expertise in object oriented programming to become a productivity tool. It strongly relies on `Spring framework` to achieve the majority of its goals. Those who are familiar with Spring's concepts like dependency injection and aspect orientation will be more comfortable here. And those who are not, will find at Helianto a fast and convenient learning path.

At last, Helianto gained an initial momentum to become an open source project when the possibilities around the "corolla" outnumbered the original group expectations. Since then, Helianto is distributed under the "Apache License v. 2.0".

Currently, Helianto has a work in progress status. This document is currently in the process of being written, so not all topics are covered. Contributions are welcome.

---

# Chapter 1. The Helianto Project

## Introduction

Helianto tries to capture common project issues and combine it with sound project practices. Many designers need to model entities like customers or suppliers, payables and receivables, and so forth. Helianto provides extensible domain and service classes for them. Many designers have a growing interest on ORM. Helianto provides basic configuration for Spring and Hibernate to achieve this, as well as the corresponding OR mapping for the supplied domain classes. Many designers would refactor the code to reduce coupling. Helianto enforces the use of dependency injection. Some have realized EJB is too complex and will find here how to keep declarative transaction management simple. Some would like to benefit from a non invasive security framework. Helianto provides configuration and a template for a secure web application using Acegi Security.

Helianto strongly relies on Spring to achieve all of this. It also embeds a couple of design decisions that narrow the power of Spring. It is a trade off. In one hand, you have to follow a pre-configured application, but in the other, you may get ready shortly.

The key values to develop Helianto classes and resources is to keep wide scope and flexibility to ease composition or extension.

In a typical use case, the Helianto user would:

- add the helianto\*.jar packages to the application classpath,
- create the presentation layer to invoke the supplied service layer,
- customize the jdbc connection through the hibernate.properties file,
- test and deploy the application. .

If the application requirements become more specific, the user would also have to create or extend domain classes, map them to the persistence layer, create or extend service facades and wire them to the dependency injection container registry.

Throughout this document, many of the aspects of such customization will be described in deeper detail. See the appendices for a brief conceptual introduction along for some of the collaborating open source packages.

## Mission

Continuously develop an application base framework that:

- is as decoupled as possible from the presentation layer, allowing the desired customization to expose service and persistence layers that fits best its purpose,
- provides an extensible and flexible service layer to accomodate both simple and complex project requirements, either being partially implemented or extended through the customization,
- concurrently provides a common domain model to solve well known business problems and a persistence layer as decoupled as possible from the datastore, and
- enforces good programming practices like design patterns usage, rich documentation and extensive testing.

## Acknowledgements

The Helianto project includes software developed by the Apache Software Foundation (<http://www.apache.org>), the Spring Framework Project (<http://www.springframework.org>) and the Ace-

gi Security System for Spring Project (<http://acegisecurity.sourceforge.net>). It also includes software from Hibernate (<http://www.hibernate.org>).

It is virtually impossible to say thank you to all the people who contributed in some way to the current state of this project. Many of them are project leaders of the packages mentioned above.

## Alternatives

As stated before, the Helianto project is not meant to be used out of the box. This is primarily because the presentation layer is not part of the project mission. Other projects, like AppFuse (<http://appfuse.dev.java.net>) or Java Application Generator (<http://jag.sourceforge.net>) provide a more comprehensive support for a full architecture.

## Software Requirements

The source code includes generics and enumerators, therefore a jdk 1.5 or higher is required.

Eclipse platform and the appropriate plugins as well as Apache Maven are recommended, although not required. Of course, there are many dependencies listed in the pom files, and Maven will help to have them ready to use.

Any sql database can be used as long as the appropriate properties are set in hibernate.properties at src/main/resources/.

## Architecture

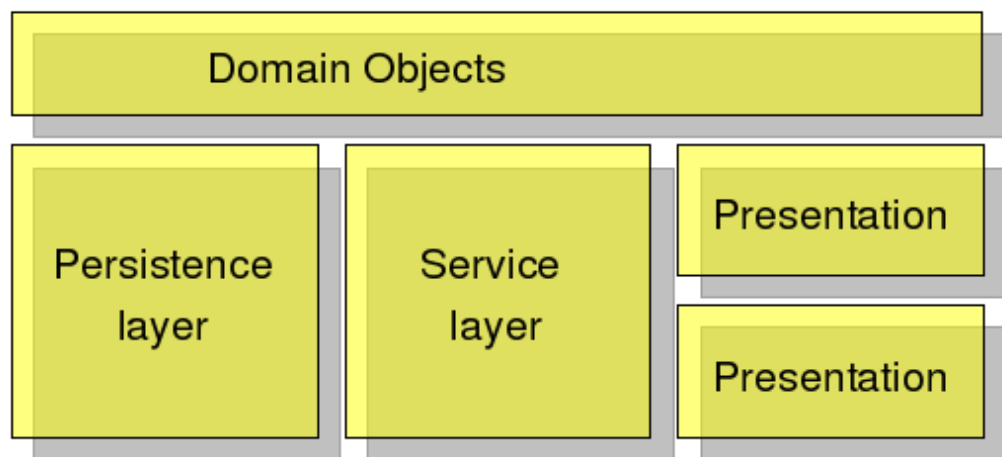


Figure 2: Architecture.

## Core functionality

## Overview

The core module has two major responsibilities:

- hold the configuration, and
- implement a basic domain model and services.

All the configuration is declarative, mostly in the form of xml files, and will be explained in the next section.

The basic domain model comprehends user and entity relationship. It is designed to be as flexible as possible to accomodate both, a single user in a small organization and multiple identities beeing shared across an internet distributed application.

The entity is a key domain object. Its responsibility is to isolate information from one business unit from another, like accounting or maintenance data. In this sense, an entity may be a whole organization, part of one organization or even a group of organizations. The application extending Helianto may choose any of such approaches.

As an immediate consequence, the domain model has finer granularity and delegates to the service layer the job of exposing simple and complex service methods. For example, there is a method to create a user and associate it by default to an entity. But there also is a method to create a credential to be supplied in a further operation that creates a user for a selected entity.

There is also a topic to be examined. It is sometimes desirable to share specifc information accross multiple entities. Customers would eventually like to share product data (at their discretion) with suppliers. Helianto is designed to create partners either as domain objects local to an entity, designated as "weak" relationships, or as entities themselves in a so called "strong" relationship. Also, weak partners can be converted to strong partners.

In order to accomplish this, helianto core module relies on several packages:

- org.helianto.core: basic domain objects,
- org.helianto.core.dao: a set of common dao interfaces,
- org.helianto.core.hibernate: an Hibernate implementation for the common dao interfaces,
- org.helianto.core.junit: extra test support,
- org.helianto.core.mail: mail notification support,
- org.helianto.core.security: Acegi Security support, and
- org.helianto.core.service: transactional core service layer,

## Basic configuration

[Pending]

## The helianto-core package

The helianto-core package plays the major role for Helianto funcionality:

- support basic Spring, Hibernate and Acegi configuration, wired togheter to provide common functionality to all other modules,
- fully implement the persistence layer, exposing a standardized dao interface,
- provide for multiple entity resolution, as entities may be organizations, personal owners, customers, suppliers, and so forth,
- implement user identity,
- allow for hierarchical authorization control, follwing the chain: supervisor -> administrator -> mod-

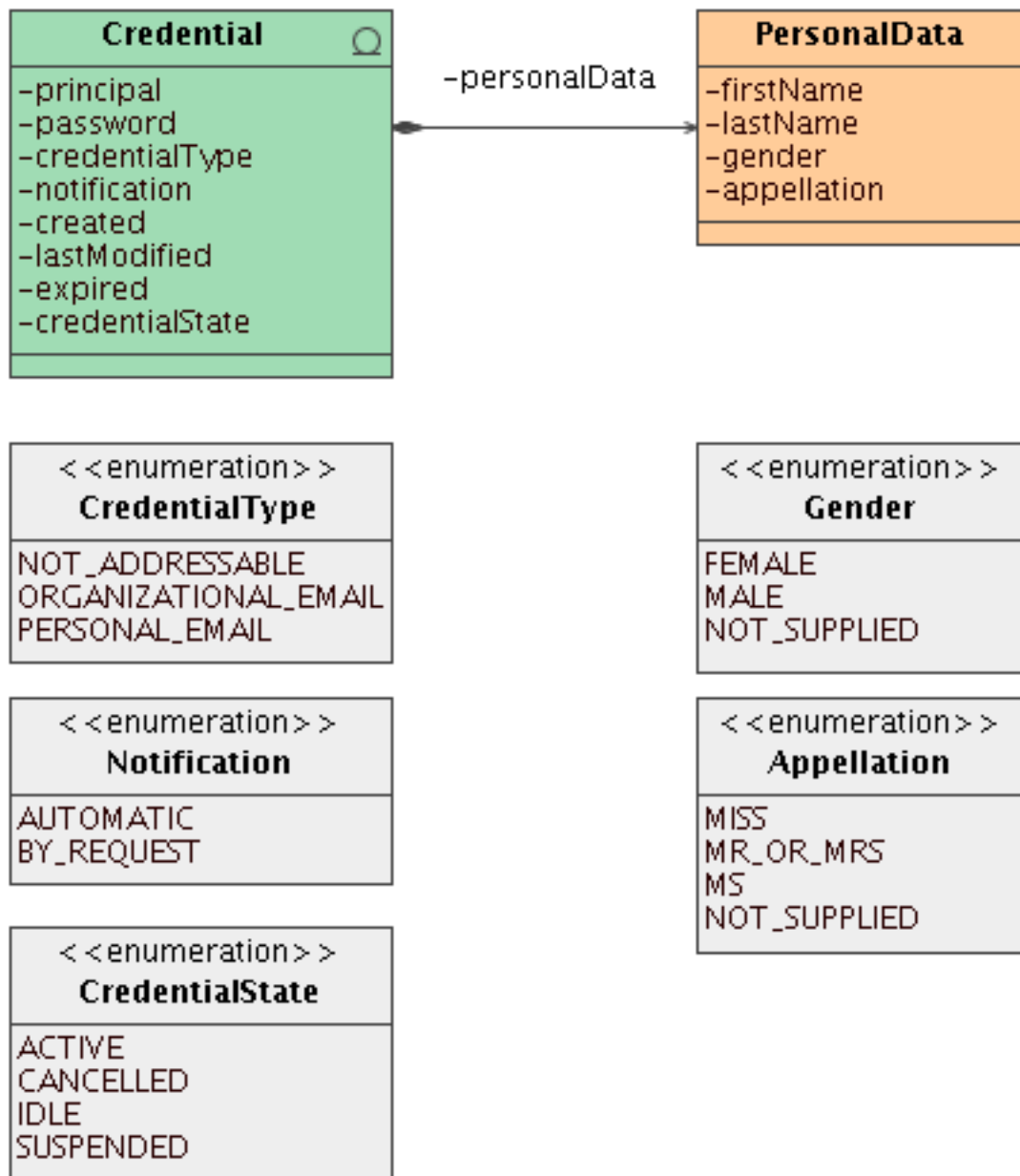


- erator -> user, and
- provide for standard event notification.

## The Credential domain object

A `Credential` provides the security system with authentication information.

A `Credential` is usually a person, but it may represent also a non-personal identity. A `Credential` is required for a person to be recognized by the system. The core domain model is designed to have a `Credential` playing different roles, even for different entities.



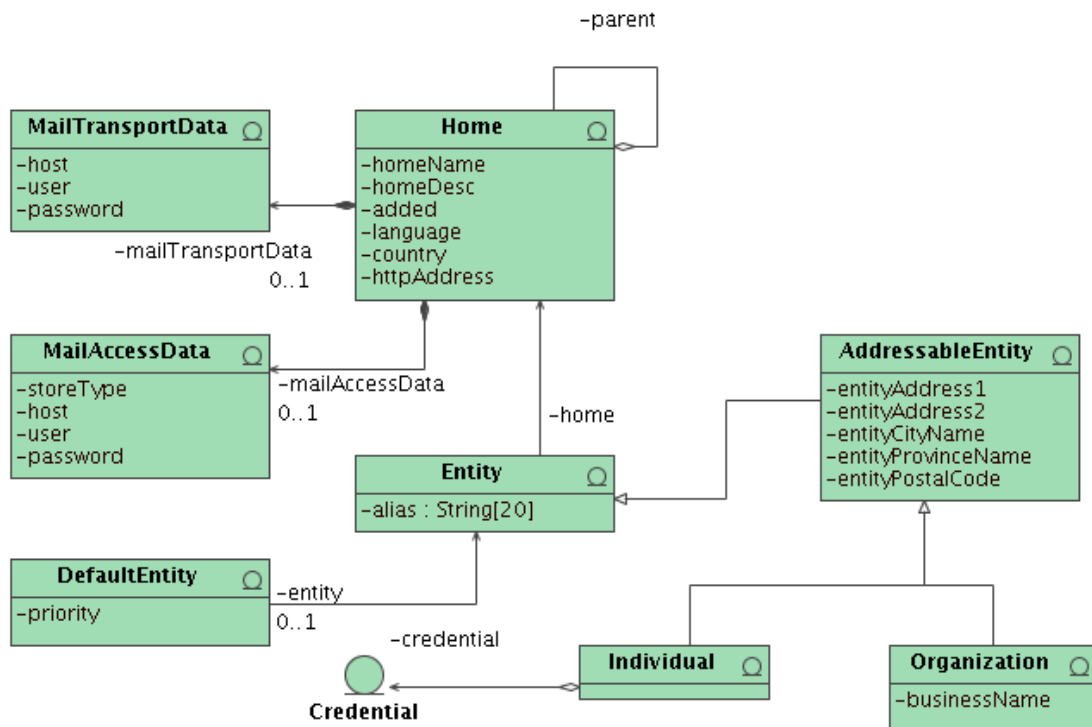
Credential and personal data.

A `Credential` must have a principal to be uniquely identified. The preferred way to supply a `Credential` with a globally unique principal is the e-mail address. A non-addressable principal may be used if the `Credential` is related to a parent having an addressable principal, or in a chain, up to at least one addressable principal.

## The Entity domain object

Helianto core functionality relies on the `Entity` abstraction. It represents the root for several hierarchies, for example, all users or all processes of an `Entity`. It may represent either an organization or part of an organization, that is in control of such hierarchies.

The `Entity` has a simple interface to allow easy extension. Its primary responsibility is to grant read-write access to the `alias : String [ 20 ]` property, which is implemented as unique in a datastore.

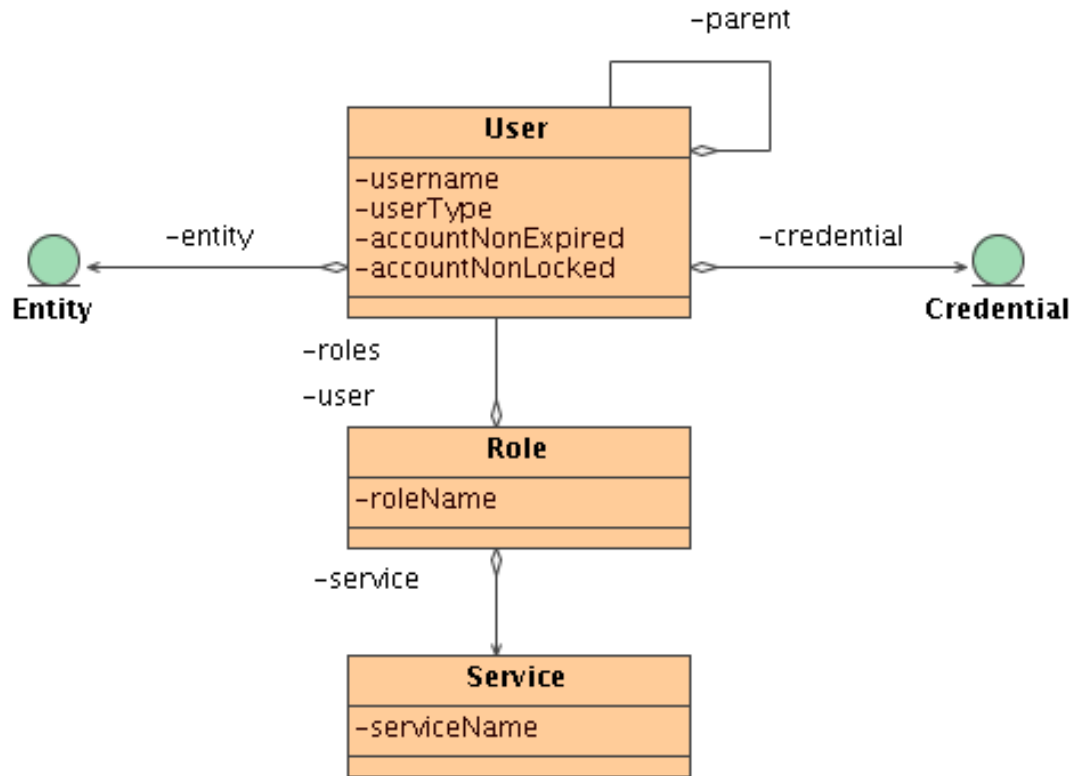


Entity model.

Each `Entity` must recognize its own `Home` property.

## The User domain object

A `User` represents the relationship between a `Credential` and an `Entity`. It is the mechanism used by a person, or any other identity, to have granted authorization to play a `Role` within an `Entity`.



User model.

## Generic DAO

[Pending]

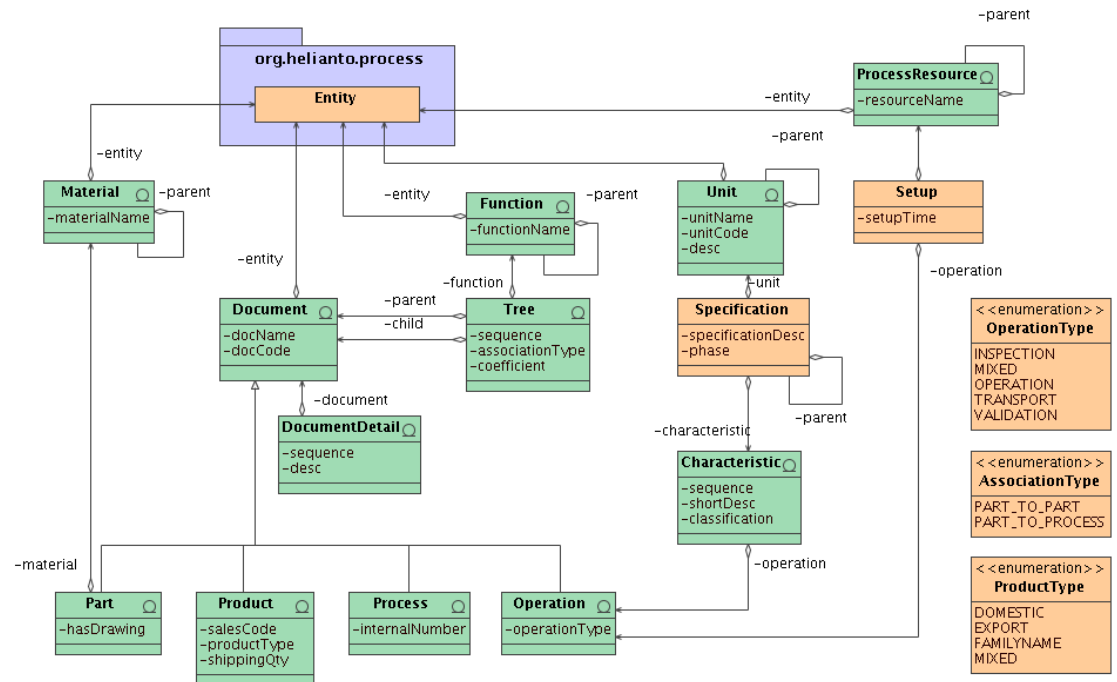
## Process functionality

### The helianto-process package

The `helianto-process` package, as the name suggests, introduces a comprehensive process approach to Helianto. Processes may be taken as organizational or production, and may be related to parts, products or other processes. The module responsibilities are:

- provide a base class, `Document`, to be root for all process components, allowing for polymorphism among parts, products, processes and operations,
- provide product and process trees, and services to maintain them,
- provide manufacturing specific domain and services,
- provide base classes for resource management.

### The Process domain model



Last updated 20051010.

[Pending]

## Sales functionality

### The helianto-sales package

The helianto-sales package is designed to provide price information to products. The module responsibilities are:

- handle, Proposals and Price list,
- provide services to calculate prices.

### The Sales domain model

[Pending]

## Finance and accountability functionality

[pending]

## A template application including access control

[pending]

---

# Chapter 2. Appendices

## Appendix A - Spring Framework and Dependency injection

First thing to say about dependency injection is that it is not invasive. It does not require your code to inherit state or behaviour from some class, or to implement any particular interface. It also means that you will not have to take things away from your code if you decide you had enough from it, and will now follow the next buzzword framework.

The action may start just after two minor adjustments around your code, or even, around legacy code:

- at design time, you choose the objects to instantiate (and how they collaborate) in a declarative fashion,
- at run time, an IoC container takes first the control of the program flow, and uses the declared configuration to actually instantiate the objects.

The application logic may then follow, with or without additional collaboration from the container.

For those who are still asking what benefits a container to jumpstart the application may bring, let's say that:

- less coupling means less plumbing or glue code, so you are free to focus on the business,
- dependencies become explicit and away of the code, a good reason to follow the well known best practice "program to an interface, not to the implementation",
- test driven development becomes easier, and
- dependency injection containers are lightweight.

Let's take an example from the Helianto configuration.

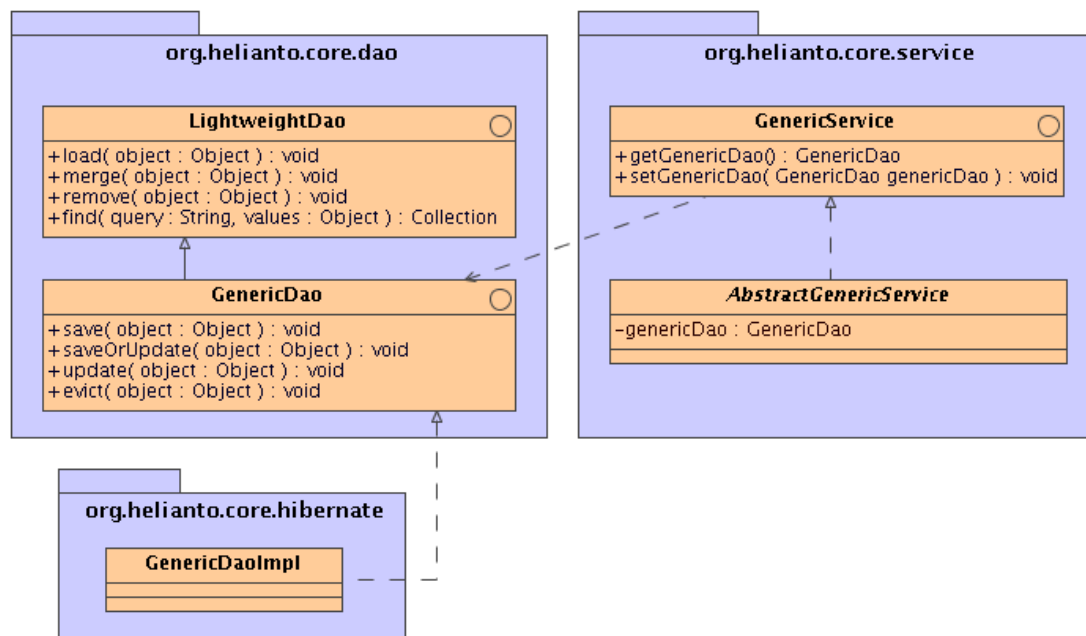


Figure 3: Dao's and dependencies.

The figure above shows the dependency between AbstractGenericService class and the interface GenericDao. Any class descending from AbstractGenericService will, of course, have the same dependency.

At the time this document was beeing written, the current design choice was to implement a concrete class for the GenericDao interface using the Hibernate orm library (see the section about object relational mapping bellow). But if there is a new better orm package to be launched in the future, or if some Helianto user decides to implement a new concrete class for GenericDao based on Oracles's Top Link orm library, a change has to be done.

The approach suggested by dependency injection is to keep the service class dependency only on the GenericDao interface and delegate to the Spring container the responsibility to "inject" the concrete class, that implements the interface, through a setter method.

Here are two code snippets to illustrate this. The first (not recommended) shows a service object with dependencies to both the interface and the concrete class. The second one relies on a setter (and a not shown dependency injection assembler) to replace the dependency on the concrete class.

```
/**
 * Example 1: contains dependencies to GenericDao and GenericDaoImpl.
 */
public abstract class AbstractNotRecommendedGenericService
    implements GenericService {

    protected GenericDao genericDao = new GenericDaoImpl();
    ...
}
```

```
/**
 * Example 2: contains dependencies only to GenericDao.
 */
public abstract class AbstractGenericService
    implements GenericService {

    protected GenericDao genericDao;

    public void setGenericDao(GenericDao genericDao) {
        this.genericDao = genericDao;
    }
    ...
}
```

To use the Spring framework terminology, the application may instantiate either a BeanFactory, an ApplicationContext or WebApplicationContext to inject, via setter methods, a dependency at run time. The job of registering the instances to be instantiated and where they are injected is easily accomplished by a xml file. As an extra advantage, neither the injected beans nor the ones which receive the injection become dependant on any Spring framework code. Here is a snippet of a xml file.

```

<!--
                Hibernate generic dao implementation
-->
<bean id="genericDao"
      class="org.helianto.core.hibernate.GenericDaoImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
...
<!--
                Core service bean
-->
...
                <bean id="coreTarget"
                    class="org.helianto.core.service.CoreMgrImpl">
                        <property name="genericDao" ref="genericDao"/>
                    </bean>
...

```

The beans in the registry can be refereced by their ids.

Now, if the programmer writes a new implementation for GenericDao, besides the new code, the CoreMgrImpl would remain untouched and the new file might contain:

```

<!--
                Toplink generic dao implementation
-->
<bean id="genericDao"
      class="org.helianto.core.toplink.GenericDaoImpl">
    ...
</bean>
...
<!--
                Core service bean
-->
...
                <bean id="coreTarget"
                    class="org.helianto.core.service.CoreMgrImpl">
                        <property name="genericDao" ref="genericDao"/>
                    </bean>
...

```

In other words, dependencies are managed in a declarative, not programatic way.

That summarizes why dependency injection became a fundamental block for the construction of the Helianto code. The Spring framework dependency injection container was the preferred implementation, not only because of its stability or consistency, but also because it offers many other advantages:

- easy integration with orm libraries,
- declarative transaction management,
- allows for aspect orientation, and as an immediate consequence, for strong but not invasive security

- management (see below),
- easy mvc integration, and also the new Webflow package.

## Appendix B - Hibernate and Object relational mapping (ORM)

Object relational frameworks' primary job is to hide the relational mechanics of SQL databases behind a graph of in memory object instances. Database records are inserted, update or deleted after object state transitions and the developer is no longer required to issue SQL statements directly.

Hibernate ([www.hibernate.org](http://www.hibernate.org)) is the current ORM choice for the Helianto project, but there is no restriction in the code to future adoptions of an alternative approach, like Oracle's Top Link or any implementation of JDO.

This appendix will explain, in short, some fundamentals of Hibernate. For further clarification, please, see the Hibernate documentation.

As SQL code is decoupled from the application code, it is possible to easily switch database vendors. Doing this, in Hibernate (and Helianto), is a matter of configuration. See the chapter "Configuration" on this guide for detailed instructions on how to change the hibernate.properties file.

Hibernate's key abstraction lies in the Session interface. A session represents a basic unit of work and should be fully understood by developers extending the `org.helianto.core.GenericDao` implementations.

A good start comes from the Session java docs:

As the above term `mapped entity` suggests, some sort of mapping is required to accomodate relational and object oriented perspectives together. Of course, only mapped entities can be made persistent.

By the mapping, in the form of a xml file, Hibernate knows how an instance is associated to a database table. Here is a sample:

```
<hibernate-mapping default-lazy="false">
  <class table="core_credential"
    name="org.helianto.core.Credential">
    <!--
                        CREDENTIAL
                        =====
    -->
    <!--
                        id
    -->
    <id name="id" type="long">
      <generator class="native"/>
    </id>
    <!--
                        principal
    -->
    <property name="principal" type="string">
      <meta attribute="use-in-equals">true</meta>
      <column name="principal" length="64" not-null="true"
        unique="true"/>
    </property>
```



```
<!--  
                                password  
-->  
<property name="password" type="string" length="20"/>  
...
```

The above listing requires two other artifacts to work with:

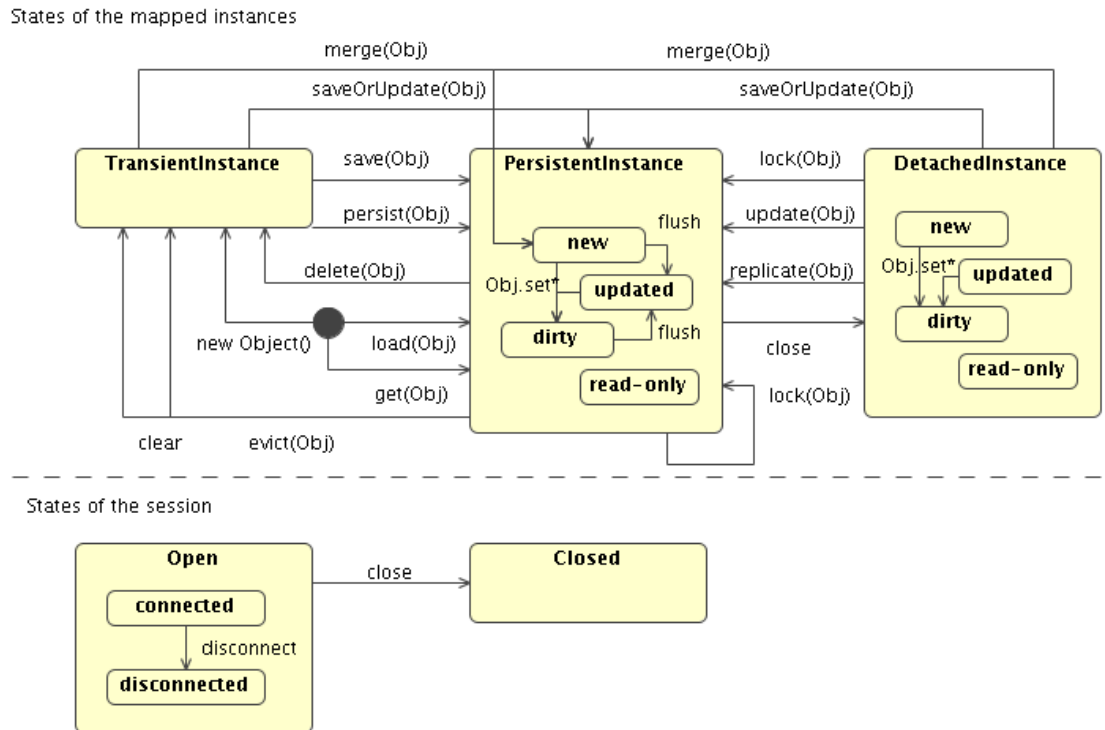
- a table named `core_credential` in the database, with fields `id`, `principal`, `password` and other (omitted),
- a serializable POJO class `org.helianto.core.Credential` with fields `id`, `principal`, `password` and other (omitted), and corresponding getters and setters following the Java Beans convention like `long getId()`, `setId(long id)`, `String getPrincipal()`, `setPrincipal(String principal)`, `String getPassword()`, `setPassword(String password)`, etc.

There are many strategies to obtain a concise set of such artifacts. The choice made in Helianto is known as `middle-out`, where the developer is responsible for the mapping file construction and the framework does its job by automatically generating the POJO's and updating the database schema accordingly.

Mapped entities instances may exist in one of three states:

- `transient`: never persistent, not associated with any Session,
- `persistent`: associated with a unique Session,
- `detached`: previously persistent, not associated with any Session.

The following state diagram tries to capture the possible transitions among such states.



Hibernate Session and Mapped Entities States.

After a session flush, the mapped entities are synchronized with database. Previous `save()` and `persist()` transitions result in an SQL INSERT, `delete()` in an SQL DELETE and `update()` or `merge()` in an SQL UPDATE.

Please, see the Hibernate documentation for further details.

## Appendix C - Acegi Security

The helianto project relies on the `Acegi-security` for Spring package to provide user authentication and authorization. The following sections have a restricted view of some main objects and concepts, while the Acegi web site documentation is comprehensive and recommended for those who look for deeper understanding.

### Security object

The security object plays a major role inside `Acegi-security` for Spring package as it coordinates the transitions required to secure an object. For the purposes of this document, the `FilterInvocation` security object is examined, and how it enables HTTP resources to be secured.

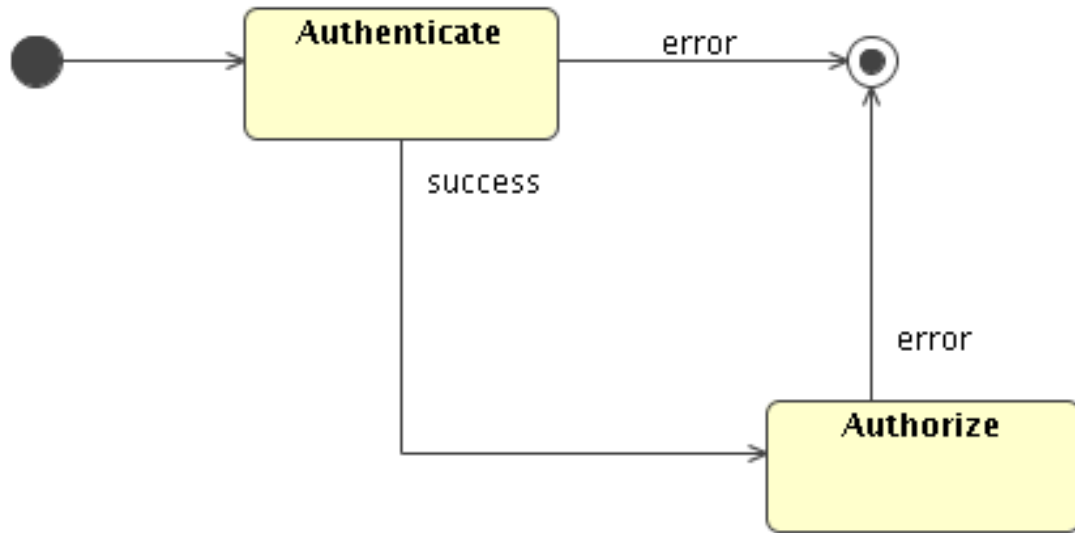


Figure 1: The security transitions.

The security interceptor has some important collaborators with distinct responsibilities:

- An `Authentication` object to hold the username, password and the authorities granted to the user.
- A `ContextHolder` which holds the `Authentication` object in a `ThreadLocal` -bound object.
- An `AuthenticationManager` to authenticate the `Authentication` object presented via the `ContextHolder`.