Helianto project

Reference Documentation

Maurício Fernandes de Castro

Helianto project: Reference Documentation

by Maurício Fernandes de Castro

Table of Contents

Preface

The Helianto Project was born after a discussion on how to leverage the competitive advantage of group of small organizations having different competences in the sotware 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 inital momentum to become an open source project when the possibilities arround 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

Developers using Helianto will find it appropriate to shorten development cycle on the server side.

Helianto tries to capture common project issues and combine it with sound project pratices. 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 code 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 quite 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 value to develop Helianto classes and resources is to keep wide scope and flexibility to ease composition or extension.

In most cases, to create an application backed by Helianto, the developer would:

- use maven to create a project and resolve dependencies,
- create the presentation layer to invoke the service layer methods available through Helianto packages,
- customize the jdbc connection through the hibernate.properties file (tables in the datastore are automatically created), and
- test and deploy the appllication to a servlet container, like Tomcat.

If the application requirements become more specific, the developer would also have to create or extend domain classes, 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

The mission is important for developers and users to keep expectations and development efforts within some basic direction.

The Helianto Project Mission

Continuously develop a server-side application base framework that:

- enforces separation of concerns,
- 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,

- 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 Acegi 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.

Software Requirements

At compile time:

- 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 (v. 2.x) are recomended, although not required. Of course, there are many dependencies listed in the pom files, and Maven will help to have them ready to use.

At run time:

- Jre 1.5 or higher is also required.
- A servlet container, like Tomcat. There is no requirement to an EJB container, although the service layer components can be easily wrapped as EJB's.
- Any sql database can be used as long as the appropriate properties are overriden in the hibernate.properties file.

Architecture

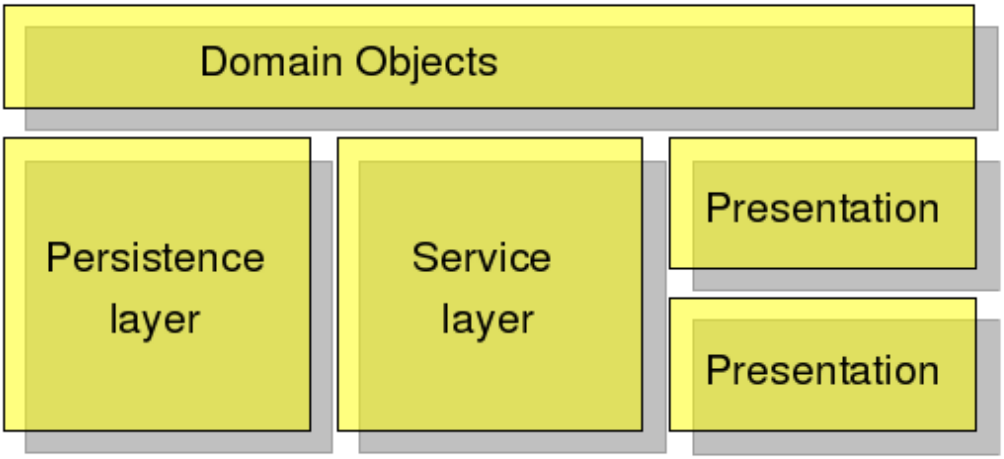


Figure 2: Architecture.

Core module

The core module has two major responsibilities:

- create the configuration required to instantiate basic persistence and service beans, 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, as follows.

Core domain classes

The org.helianto.core.Entity is an Helianto top abstraction and a key domain object. Its responsibility is to isolate information from external parties. An entity may represent a whole organization or a single person. In other words, if accounting or maintenance data from one organization should not be freely accessed by a second organization or person, they must be distinguished by entity.

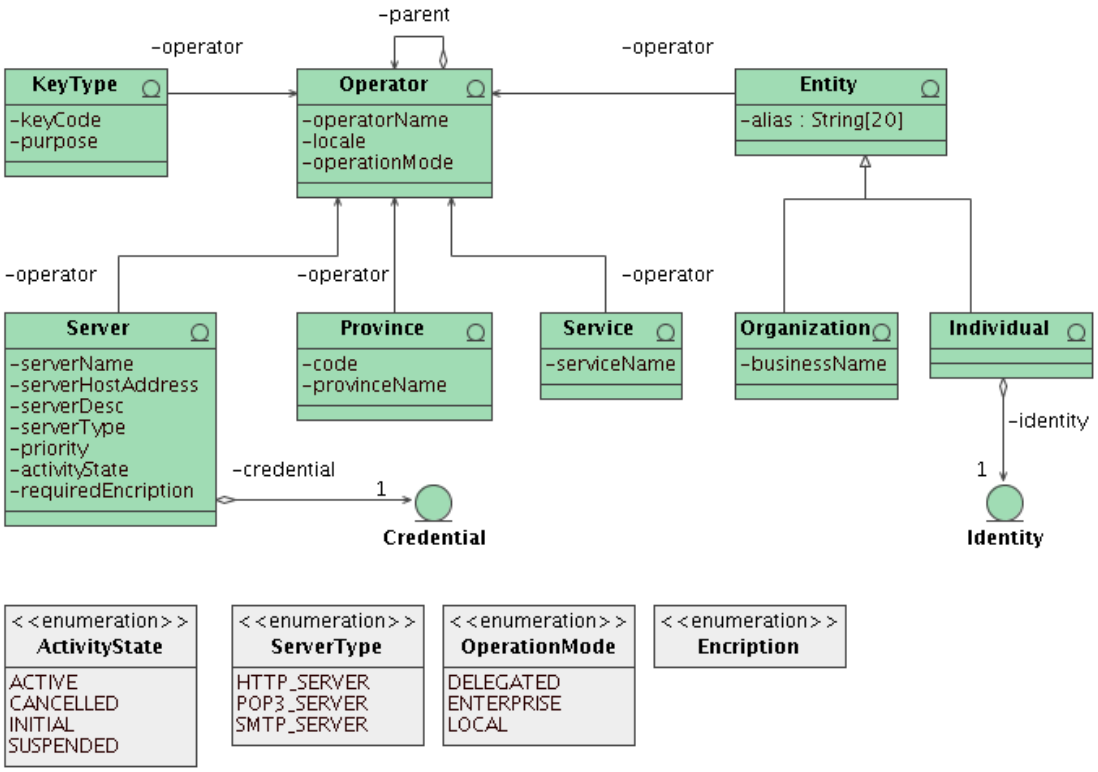
Please notice that org.helianto.core.Entity is completely different from the javax.persistence.Entity used in the persistence layer. In future releases, the class org.helianto.core.Entity may be renamed to avoid confusion.

The key to access an entity is an unique alias. A second distinction level, introduced by the class org.helianto.core.Operator, is avaliable to provide scalability. As a consequence, the entity alias should not be repeated within the group of entities controlled by an operator.

The need for such an operator abstraction is easier to understand when a single datastore is required to hold data with different key characteristics, like language, country or access to services. For applications limited to small organization boundaries (obviously sharing the same language, etc.), Helianto offers ap-

propriate service layer methods that hide the operator.

In both cases, for large or small datastores, the operator acts as a bridge between entities and top level features.



Entity model.

Next important domain abstraction is represented by the org.helianto.core.User class. First thing to note is that the every user belongs to a single entity. It means that the user class has a local reference to the owning entity, or, if you prefer the relational model, the user table has a foreign key to associate it to the entity table.

The user main responsibility is to hold authorization data. Thus, authorization is granted as a per entity basis. To allow one individual to access services in several entities, the user class delegates authentication responsibilities to a couple of distinct domain classes: org.helianto.core.Identity and org.helianto.core.Credential.

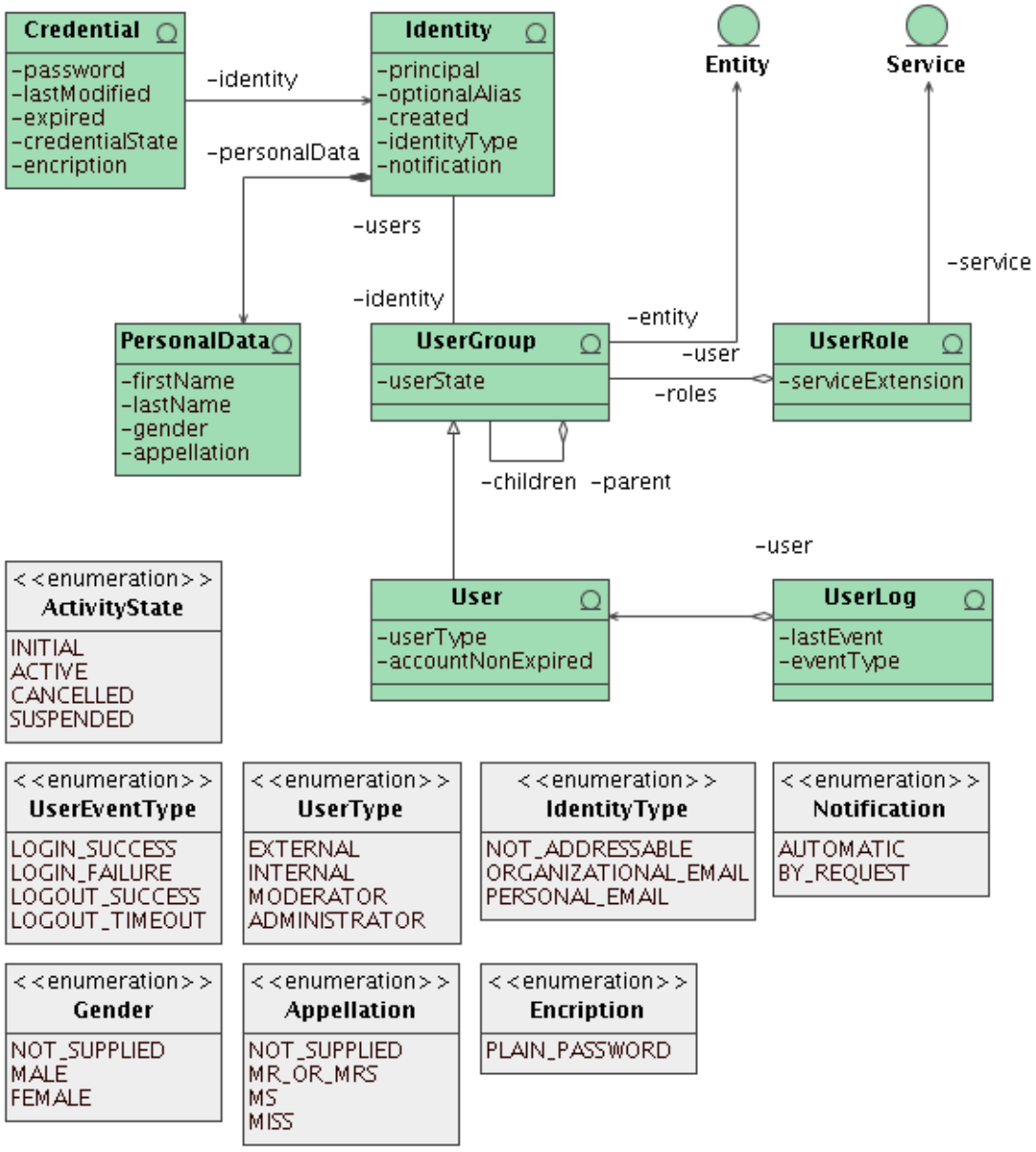
The Acegi Security package approaches authorization and authentication requirements in a very mature way (see appendix C). The org.helianto.core.security.UserDetailsServiceImpl class makes extensive use of Acegi's interfaces and classes to securely place the authenticated user instance at reach from any application layer.

So far, the Helianto built-in support to obtain an authenticated user is a simple solution to many core application issues:

- the user aggregates a set of user roles and the corresponding keys to registered services (see org.helianto.core.UserRole and org.helianto.core.Service) allowing the customization of user authorization,
- the user aggregates an identity and his/her personal data,
- the user has a reference to the owning entity, exposing a parameter appropriate to select other do-

main classes, like accounts, customers, resources, etc.,

- the owning entity, available through the user, gives access to basic parameters, like language, locale specific date and time format, a set of regional parameters as provinces (see org.helianto.core.Province) and applicable keys (see org.helianto.core.KeyType), as well as mail transport or store server configuration (see org.helianto.core.Server),
- user and its superclass, org.helianto.core.UserGroup, can be associated to each other to create an hierarchical authorization model.



Authorization model.

Basic configuration

The core module also contains basic configuration files:

- dataSource.xml: sets an Apache DBCP connection pool up, according to properties defined in the file hibernate.properties,
- sessionFactory.xml: sets an Hibernate Session Factory up, using Spring Framework classes, and defines conventions to both Hibernate mappings location and JPA annotated classes configuration,
- transaction.xml: sets a transaction manager and a manager template up, in order to ease the declarative transaction management feature of the Spring Framework,
- security.xml: wires up the Acegi Security classes required to work with the domain model described above,
- support.xml: several message related beans,
- core.xml: data access objects and service managers for the core domain model.

All files above conform to the spring-beans.dtd, i.e., they define how a Spring Framework context will be created. By convention, they are all located at the library classpath unde the "deploy/" folder. Please, check the Appendix B for a short overview of the dependency injection concept.

Two other (already mentioned) Hibernate specific configuration files deserve a closer look:

- hibernate.properties: sets the jdbc driver, connection url, hibernate dialect, user and password and other optional parameters required to connect to a database,
- hibernate.cfg.xml: declares JPA annotated classes.

The default /hibernate.properties file (see below) must be overriden in the final application classpath root to customize the database connection. Please notice that the hibernate.hbm2ddl.auto=update configuration is only required to create and update the database structure just after the application installation and can be commented out to improve performance.

```
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.connection.url=jdbc:hsqldb:file:target/testdb/db;shutdown=true
hibernate.dialect=org.hibernate.dialect.HSQLDialect

hibernate.max_fetch_depth=3

hibernate.hbm2ddl.auto=update
hibernate.show_sql=true
hibernate.connection.pool_size=3

hibernate.connection.username=sa
hibernate.connection.password=
```

The persistence layer

[Pending]

Security

Security can be activated in the presentation layer, but also in the service layer thanks to the `Acegi Security for Spring` package (see appendix for details). Not much as implementing an interface is necessary to satisfy Acegi's programming requirements. All the rest can be done in a declarative fashion. Helianto has such implementation and a couple of declared configurations ready.

The class `org.helianto.core.security.UserDetailsServiceImpl` implements the interface `org.acegisecurity.userdetails.UserDetailsService`. Just after the user cre-

dentials are supplied, this class looks in the datastore for a matching `User`. One of three alternatives is followed:

- username and password do not match and are rejected,
- as one single credential (username and password) may be connected to more than one entity, the entity selection is taken from the last choice registerd in an `UserLog` domain object,
- if there is no `UserLog` yet, the `UserDetailsServiceImpl` tries to guess an `Entity`.

[Pending]

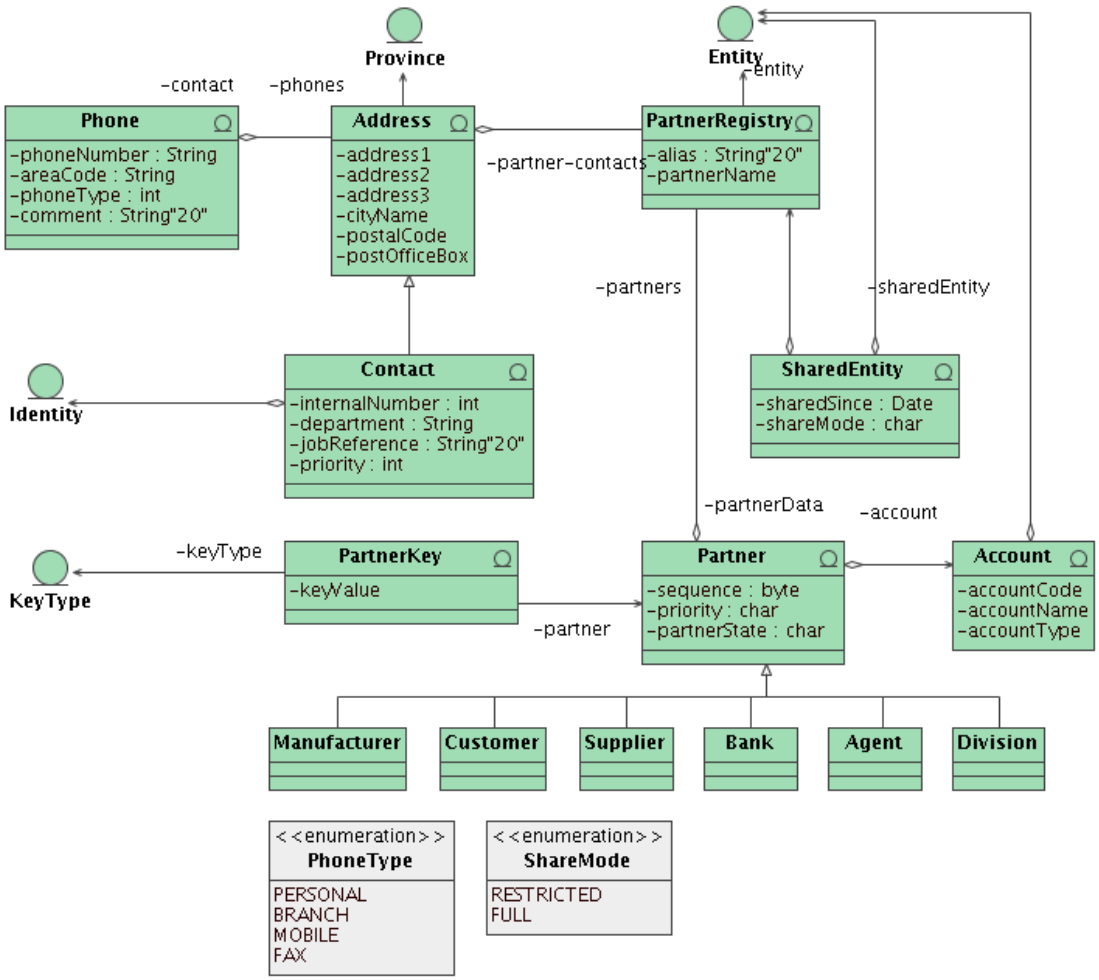
Partner module

The partner module introduces customers, suppliers and other organization related third parties.

Partners start to exist to some entity just after a simple registration. The key to access a org.helianto.partner.PartnerRegistry is a partner alias string, unique within the entity.

[Pending]

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.



Partner model.

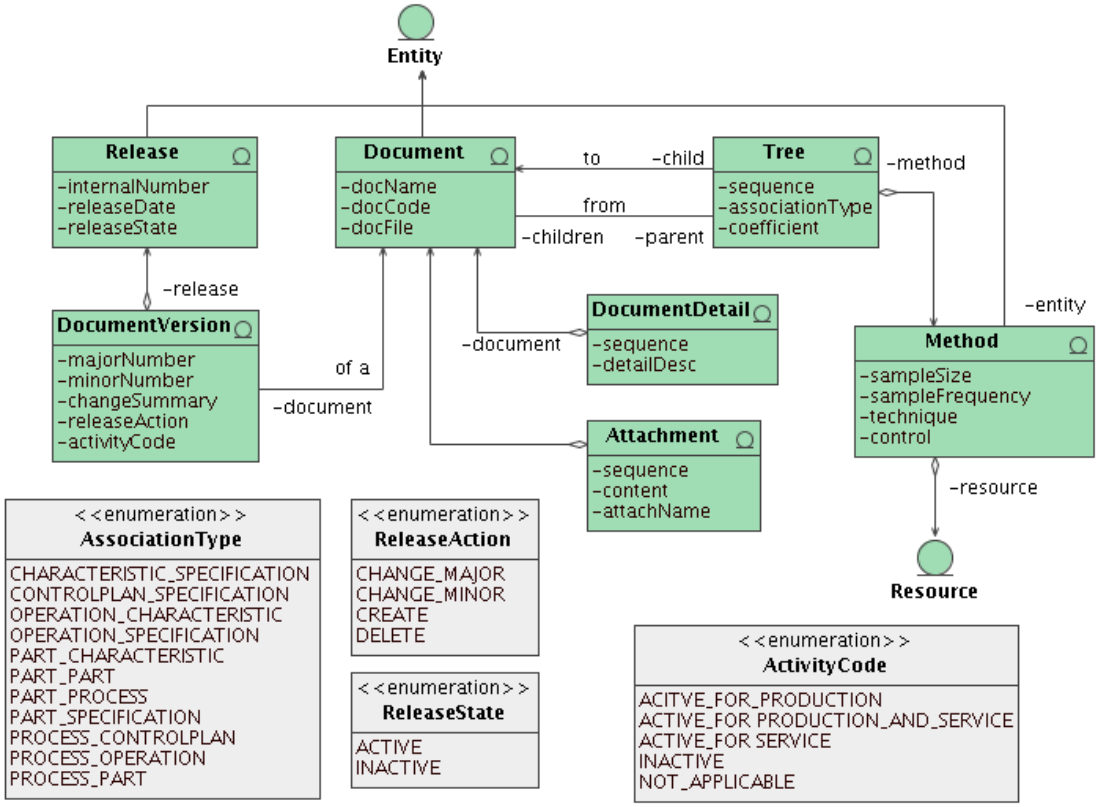
Process module

The `helianto-process` package

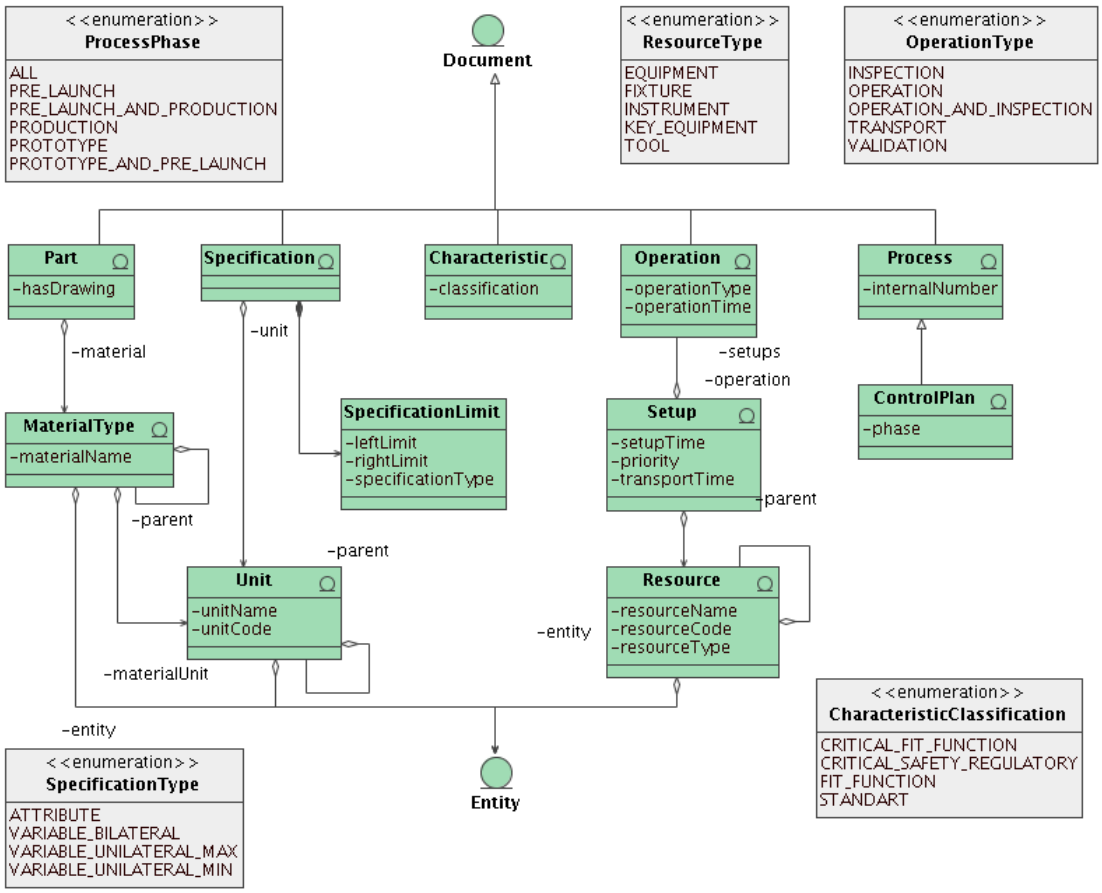
The `helianto-process` package, as the name suggests, introduces a comprehensive process approach to Helianto. Processes may be taken as organizational or production, and may be related to parts, products or other processes. The module responsibilities are:

- provide a base class, `Document` , to be root for all process components, allowing for polimorphism 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



Document association.



Document hierarchy.

[Pending]

Product module

The `helianto-product` package

The `helianto-product` package is designed to provide product and price information to processes. The module responsibilities are:

- handle, `Proposal` s and `Price` list,
- provide services to calculate prices.

The Product domain model

[Pending]

Finance and accountability module

[pending]

A template application including access control

[pending]

Chapter 2. Apendices

Appendix A - 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) 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` implemetations.

A good start comes from the Session java docs:

> The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes.

As the above term `mapped entity` suggests, some sort of mapping is required to accomodate relational and object oriented perspectives togheter. 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 artifacs to work with:

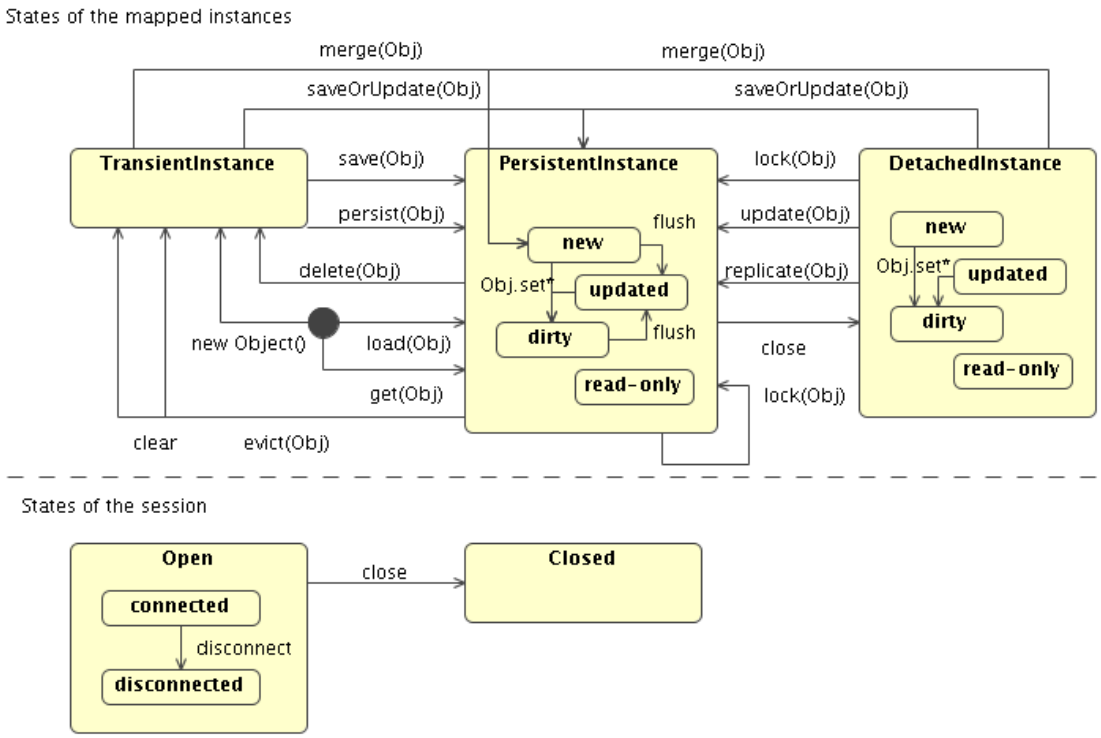
- a table named `core_credential` in the database, with fields `id` , `principal` , `password` and other (ommited),
- a serializable POJO class `org.helianto.core.Credential` with fields `id` , `principal` , `password` and other (ommited), and corresponding getters and setters following the Java Beans 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 B - 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 wil 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 arround your code, or even, arround 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,

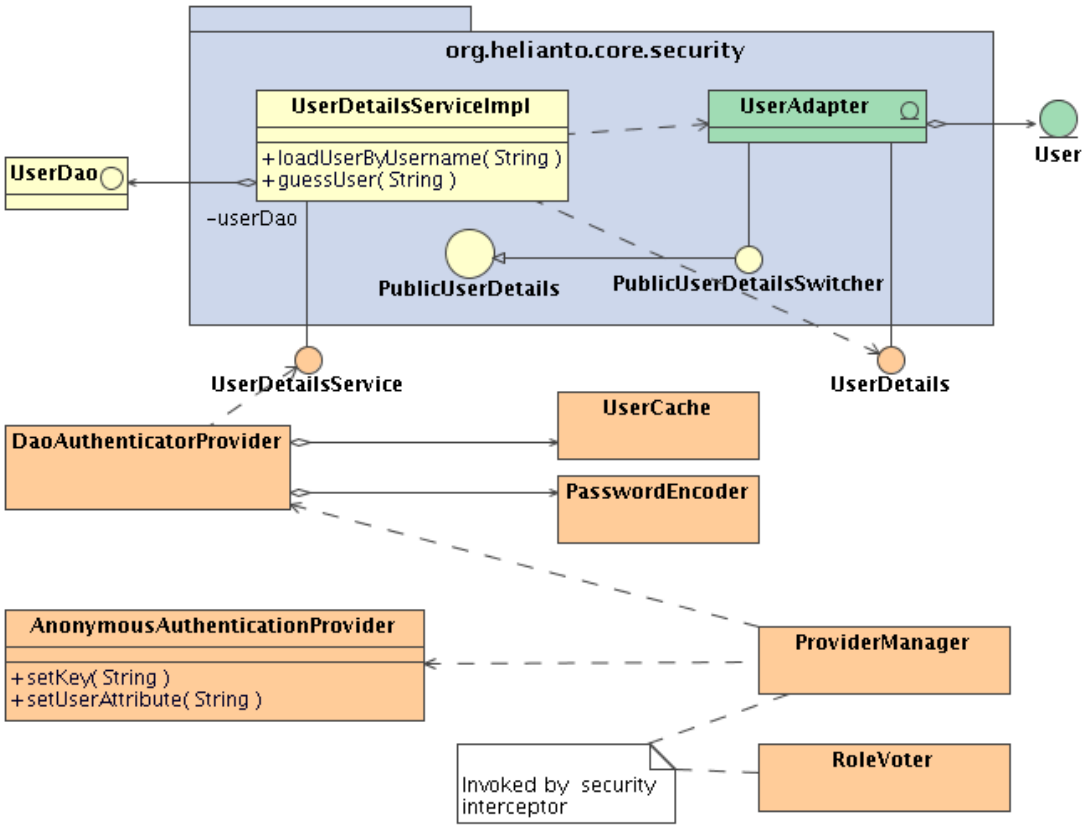
- dependencies become explicit and away of the code, a good reason to follow the well known best pratice "program to an interface, not to the implementation",
- test driven development becomes easier, and
- dependency injection containers are lightweight.

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 `Filter-Invocation` security object is examined, and how it enables HTTP resources to be secured.



Acegi configuration.

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` .