

Mod4j Application Architecture

Eric Jan Malotaux

Mod4j Application Architecture

Eric Jan Malotaux

1.2.0

Copyright © 2008-2009

Table of Contents

1. Introduction	1
1.1. Purpose	1
1.2. References	1
1.3. Abbreviations	1
1.4. Structure of this document	2
2. Logical View	3
2.1. Overview	3
2.2. Presentation Layer	3
2.3. Service Layer	4
2.3.1. Data Transfer Objects	4
2.4. Business Layer	5
2.5. Data Layer	5
2.6. Domain Model	5
3. Implementation View	7
3.1. Modules	7
3.2. Packages	7
3.3. Spring	7
3.4. Hibernate	8
3.4.1. Mapping	8
3.4.2. Access strategy	8
3.4.3. Optimistic concurrency control	9
3.4.4. Spring support	9
3.4.5. Enumerations	9
3.4.6. Primary keys	10
3.5. Domain Model	10
3.5.1. Constructors	10
3.5.2. equals and hashCode	11
3.5.3. Business rules	11

List of Figures

2.1. Logical view	3
-------------------------	---

List of Tables

3.1. Packages	7
---------------------	---

List of Examples

3.1. Optimistic concurrency control mapping	9
3.2. Optimistic concurrency control Java field	9
3.3. Enumeration mapping	9
3.4. enum Java implementation	9

Chapter 1. Introduction

1.1. Purpose

The purpose of this document is to describe the architecture of web applications generated by Mod4j. Users of Mod4j need to understand this in order to be able to extend the generated application.

1.2. References

Bibliography

- [Busch96] *Pattern-Oriented Software Architecture - A System of Patterns*. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. John Wiley & Sons. 1996.
- [Evans03] *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Eric Evans. Addison-Wesley. 2003. 0-321-12521-5.
- [Fowler02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley. 2002. 0-321-12742-0.
- [Spring25] Rod Johnson et al.. *java/j2ee Application Framework*. The Spring Framework - Reference Documentation. Version 2.5. Copyright © 2004-2007.

1.3. Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
CRUD	Create, Read, Update, Delete
DSL	Domain-Specific Language
DRY	Don't repeat yourself
DTO	Data Transfer Object
EJB	Enterprise JavaBean
JDBC	Java Database Connectivity
JDK	Java Development Kit
MDD	Model-Driven Development
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
REST	Representational State Transfer
RMI	Remote Method Invocation
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol

SQL

Structured Query Language

1.4. Structure of this document

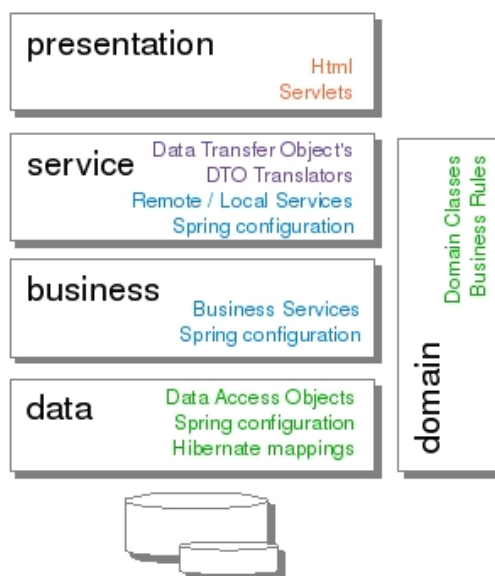
Chapter 2 describes the logical structure of an application generated by Mod4j , while chapter 3 describes its technical structure.

Chapter 2. Logical View

2.1. Overview

This chapter describes the logical composition of the Mod4j Reference Architecture. It defines the layers, the component types in each layer and the responsibilities of these component types.

Figure 2.1. Logical view



The *Layers* architectural pattern itself is described in [Busch96]. The *Presentation Layer* provides an interface to the system to a human user, presenting information and interpreting actions of the user. It uses the *Service Layer* to obtain the information or execute other operations that the *Service Layer* offers. The *Service Layer* provides access to the services of the *Business Layer* using a particular access protocol, like *SOAP*, *EJB*, *RMI* or local method invocation. The *Business Layer* provides services that implement functionality and business logic. The *Data Layer* provides access to persistent data on behalf of the *Business Layer*. The *Domain Model* consists of classes that model the part of the world that the application is about. The next paragraphs describes the responsibilities and the structure of the layers in more detail.

2.2. Presentation Layer

The *Presentation Layer* contains components that present information to a user and interprets his actions by executing functions of the system. Typical responsibilities of the *Presentation Layer* include:

- Presenting data to the user. This implies mapping fields on a screen to attributes of *Data Transfer Objects* received from the *Service Layer*, formatting of data, internationalisation en localisation of data.
- Accepting data entered by the user. This implies applying simple validations on the data entered and keeping track of this data.

This *Presentation Layer* should not execute business functionality by itself, but instead delegate that responsibility to the *Service Layer* by calling its operations. Data is transferred to and from the Ser-

vice Layer in the form of *Data Transfer Objects*.

TODO: expand this section when the User Interface *DSL* is nearer to completion.

2.3. Service Layer

The primary responsibility of the *Service Layer* is to isolate the business layer from the various protocols in use for remotely invoking its service operations. Examples of such protocols are *SOAP* or *RESTful* Web Services, messaging, *EJB*, *RMI* or plain old local method calls. A service layer presents the interface of the business layer in the form required for one particular access protocol. Therefore, a separate service layer is needed for each protocol by which the system will be accessed. This makes it clear that the presentation layer should not contain any business functionality, because that would have to be duplicated for each separate service layer. Code duplication is bad: it violates the *DRY* (Don't Repeat Yourself) principle.¹

In addition, the service layer is responsible for:

- authentication and authorisation of the calling identities
- exposing a collection of *Data Transfer Objects* to be used for transferring data between the service layer and its callers
- translation of the information in the data transfer objects used for communication with the presentation layer to and from *Domain Model* objects used for communication with the business layer
- making the call to each service operation an atomic (*ACID*) transaction.

2.3.1. Data Transfer Objects

The purpose of Data Transfer Objects (*DTO*'s) is to decouple the core of the application from client applications, including the user interface, that communicate with it. A collection of *DTO*'s form a common vocabulary between the communicating applications. In a *SOA* environment for instance, an application will commonly be required to communicate with other applications using a common *canonical datamodel*, that is defined outside of the control of the application we have to develop.

DTO's are very similar in structure to domain model classes. They have properties and can have references to other *DTO*'s. They do not have validators and no constructors with mandatory fields. All properties are nullable.

For each *DTO* there is a corresponding *Translator* class. A *Translator* defines two methods: `toDto` and `fromDto`. The `toDto` method translates a domain model object into a *DTO* that is based on it. When a *DTO* has references to other *DTO*'s that corresponds to references of the domain model class, those objects are also translated, recursively. The translators operate within a transaction and therefore also on objects connected to the same *Hibernate* session. That means that while traversing associated domain object to translate into their corresponding *DTO*'s, the translator may cause *Hibernate lazy loading* to retrieve objects from the database automatically.

The `fromDto` method translates the *DTO* into an instance of its corresponding domain model class. If the *DTO* has references however, these are *not* translated. There is a specific reason for this asymmetric behaviour that has to do with passing *DTO*'s corresponding to existing domain objects as an arguments to services. In general, the client can manipulate objects referenced by a *DTO* at will, adding, deleting or updating instances. Faithfully merging these intended changes with an existing object retrieved from persistent storage would require the translator to interpret the changes by comparing them to the state of the corresponding domain object, including the referenced associated objects. This is at the moment considered too great a challenge. A client of the application will have to send a *DTO* and its associated objects by sending the *DTO* itself, and each of the deleted, changed or added associated objects by calling separate specialised services that reflect the intended behaviour.

¹See for example: http://en.wikipedia.org/wiki/Don't_repeat_yourself

2.4. Business Layer

The *Business Layer* is a *facade* for the *Domain Model*. It exposes the business functionality of the application in the form of methods, with arguments and return values from the domain model. These methods:

- persist and retrieve domain objects by delegating to the *Data Layer*
- delegate to domain model objects to execute reusable business functionality
- implement business logic specific to a single business process

2.5. Data Layer

The responsibility of the *Data Layer* is to persist and retrieve data. The data is transferred between the data layer and its calling business layer in the form of arguments and return values based on the domain model. The persistence logic is dependent on the particular persistence technology in use, like a relational database, textfiles, XML documents or an XML database, or another application that exposes services for accessing data.

The data layer contains a number of data access objects, in principle one for each domain model class. Each such object:

- Has *CRUD*-methods to create (C), retrieve (R), update (U) or delete (D) data contained in one domain model object
- Can have methods to retrieve a list of objects based on a query
- Must transform data from the domain model to the format in which they are persisted, and vice versa. This is done with *Hibernate* a Object-Relational Mapping (*ORM*) framework, like or *Toplink*.
- Can offer paging facilities when a large result set may be retrieved.
- May not maintain state.

2.6. Domain Model

The *Domain Model* contains classes that model the part of the world that the application is about. The domain model is developed in an object oriented way as advocated by [Fowler02] and [Evans03]. As much business functionality as possible is implemented as methods on domain model classes. Domain model objects:

- Should expose methods that perform business actions or complex computations.
- Should not be aware of persistence-related issues. Domain model objects should not have to worry about whether or not they are persisted, and how. For instance: they should not have create, retrieve, update or delete methods.
- Trigger validation of business rules at any state change. Business rule violations are reported by raising an exception. When more than one business rule is violated by a state change, only one exception is raised that contains information about each one.
- May leave the validation of certain types of business rule to lower layers, like the database. Good candidates for instance are unique rules, that are easily enforced by a relational database by defining unique constraints. The downside of this approach is that business rule violations in these cases are raised at a different moment than ordinary business, and that they have to be treated dif-

ferently.

- May trigger events to notify other interested objects of state changes.

Chapter 3. Implementation View

This chapter describes how the architectural layers described in the previous chapter are implemented. It describes how the code is divided into modules, how the Java classes are organised in packages, which frameworks are being used and in what way.

3.1. Modules

The application is implemented as a *Maven* project, consisting of a *parent* project of type *pom*, and a number of modules, one for each layer of the Logical View. The modules are contained in subdirectories of the directory of the parent project.

The names of the modules are formed by concatenating the Mod4j Project Name, a dash ("-") and a keyword derived from the architectural layer the module implements. For a hypothetical Mod4j project "RecordShop", the module names would be: RecordShop-service, RecordShop-business, RecordShop-data and RecordShop-domain.

The Maven project can be built by Maven from the command-line. Alternatively, the modules may be imported into Eclipse with the *Q4E* -plugin, and built from within Eclipse.

3.2. Packages

The Java classes are organised in Java packages. The package names are prefixed with a root package name provided by the user at the time the Mod4j project was created by the Mod4j Project Creation Wizard. After project creation the root package name is stored in the file `src/model/mod4j.properties` as the value of the property `rootPackage`. This file can be edited manually to change the root package name. The packages names below this common prefix start with a package name corresponding with the component name, that in turn is derived from the architectural layer that it implements.

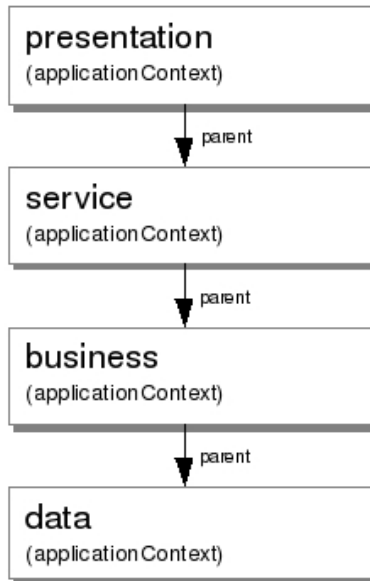
Table 3.1. Packages

Name	Description
service	Interfaces and implementations of the service layer.
service.dto	Data Transfer Objects.
service.dto.translators	Java classes that translate Domain Objects to Data Transfer Objects, and vice versa.
business	Interfaces and implementations of the Domain Services in the business layer.
data	Interfaces that define the contracts of the Data Access Objects.
data.hibernate.spring	Spring/Hibernate specific implementations of the Data Access Object interfaces in the data package.
domain	Domain Objects: <i>POJO</i> classes that implement the Domain Model.
domain.businessrules	Java classes that implement Business Rules.

3.3. Spring

The Spring framework is used as a general application framework. The objects making up the application are being wired up by the Spring framework based on a number of XML configuration

files, generally one per layer. The Spring `ContextSingletonBeanFactoryLocator` class is used to create multiple hierarchical Spring IoC container instances, also called bean factory or application context in Spring parlance, from these XML configuration files. For the container of each layer, the container of the lower layer is the parent container.



Beans defined in a direct or indirect parent context are visible in the child context, but not vice versa. This hierarchical container is then used as the parent container of a web application container. In this way, containers and the beans contained in them, may be shared between several web applications, and loaded only once. Moreover, the hierarchical composition of the container guarantees that dependencies between them are parallel to the dependencies between the layers of the logical view. See section 3.9 *Glue code and the evil singleton* in [Spring25] and the Javadoc for `ContextSingletonBeanFactoryLocator` for more details.

3.4. Hibernate

Hibernate is an object-relational mapping framework used to implement the data layer. This section documents the choices made in several areas of this framework.

3.4.1. Mapping

Hibernate offers a choice between mapping using *JDK 5.0* annotations inside the domain model classes being mapped, and mapping using one or more separate XML mapping documents. We choose using XML mapping documents. The rationale is that we want to keep persistence concerns strictly separate from pure business logic. This way we can more easily change mapping strategies or even persistence frameworks.

Each persistent class is mapped in a separate XML document located in a directory corresponding to the fully qualified Java package of the class being mapped, and with a name constructed from the Java class name and a suffix ".hbm.xml". So, the fully qualified path relative to the package root, for an XML mapping document for a persistent class named `org.company.recordshop.domain.Artist` would be `org/company/recordshop/domain/Artist.hbm.xml`.

3.4.2. Access strategy

Hibernate offers a choice between two strategies to access the fields of objects: *property access*, which is the default, and *field access*. We choose the second, *field access*. We prefer to keep programmatic access to the state of the object separate from framework access. For example, using a setter will trigger data validation, which is clearly not necessary when populating an object from persistent store. And it might trigger data transformations for derived fields.

3.4.3. Optimistic concurrency control

Hibernate offers several strategies to implement concurrency control. We choose optimistic concurrency control based on a version property, with a negative unsaved-value. In the persisted class, this is a private int field named `version`, initialized to -1. The following example from a Hibernate mapping file illustrates this:

Example 3.1. Optimistic concurrency control mapping

```
<version name="version" unsaved-value="negative"/>
```

The corresponding field declaration in the mapped Java class looks like this:

Example 3.2. Optimistic concurrency control Java field

```
@SuppressWarnings("unused")
private int version = -1;
```

Since this field is only used by Hibernate, there are no access methods for it.

3.4.4. Spring support

The Java implementations of the data access objects are based on the Spring `HibernateDaoSupport` class and its `HibernateTemplate` helper class. This implies that all `Hibernate`, `SQL` and `JDBC` exceptions will be converted to an appropriate subclass of `DataAccessException` class.

3.4.5. Enumerations

Java enum's are mapped with a specific implementation `<rootPackage>.data.GenericEnumUserType` of the Hibernate interface `org.hibernate.usertype.UserType`. The following example illustrates this:

Example 3.3. Enumeration mapping

```
<property name="sexe">
  <type name="org.company.recordshop.data.GenericEnumUserType">
    <param name="enumClass">
      org.company.recordshop.domain.SexeEnum
    </param>
  </type>
</property>
```

The `enumClass` must be a Java enum class with a method `id()` and a method `value(Integer id)`. When an object with a enumeration property is persisted, the return value of the `id` is stored in the database. When an object is loaded from the database, the enumeration instance returned by the method `value(Integer id)` with the persisted value as argument is set as the value for the enumeration property.

Example 3.4. enum Java implementation

```
package org.company.recordshop.domain;
```

```
import java.util.EnumSet;
import java.util.HashMap;
import java.util.Map;

/**
 * Sexe enumeration.
 */
public enum SexeEnum {

    FEMALE(1), MALE(2);

    private static final Map<Integer, SexeEnum> lookup =
        new HashMap<Integer, SexeEnum>();
    static {
        for (SexeEnum s : EnumSet.allOf(SexeEnum.class))
            lookup.put(s.id(), s);
    }

    private Integer id;

    private SexeEnum(Integer id) {
        this.id = id;
    }

    public Integer id() {
        return id;
    }

    public static SexeEnum value(Integer id) {
        return lookup.get(id);
    }
}
```

3.4.6. Primary keys

Primary keys are instances of `java.lang.Long`. This field may be modified only by Hibernate and therefore there are no methods that modify this field. A getter may be useful.

3.5. Domain Model

The domain model classes need very little framework support. They are implemented as Plain Old Java Objects (POJO's). The business rule implementations do rely on some classes of the Spring framework, but they are implemented as separate classes. The domain model classes are not in any way dependent on them.

Domain model objects are designed to be at all times in a consistent state. To be in a consistent state means that all mandatory properties have a value, and that the object satisfies all business rules defined for the class. This principle is enforced by providing

- a constructor with all mandatory properties as parameters
- a set of classes that implement business rules, including rules to check for the presence of non-null values for mandatory properties.

The business rules must be checked whenever the state of an object changes. For example, at the end of the constructor with the mandatory properties, and at the end of every setter.

3.5.1. Constructors

Every domain model class needs at least two constructors. One of these is a no-argument constructor that is used by Hibernate. This constructor is not intended to be used for any other purpose, so it

must be declared with the least visibility that still allows it to be used by Hibernate. This is protected visibility. This constructor must be present, but does not have to do anything. The other constructor takes all mandatory properties as parameters. The actual argument values for these parameters must be validated using the appropriate business rules.

3.5.2. equals and hashCode

All domain model classes must override the standard methods `equals` and `hashCode` in a consistent way. Two domain model class instances are considered equal either if they are the same objects, or if they belong to the same class *and* have the same value for their `id` property. The `hashCode` could return the `hashCode` of the `id` property if it is non-null, and the `hashCode` of the superclass if it is null.

3.5.3. Business rules

Business rules are implemented as separate classes, one class per business rule. The business rule implementation are based on two interfaces from the Spring framework: `Validator` and `Errors` as described in the paragraph 5.2 Validation using Spring's `Validator` interface of [Spring25]. A difference between our approach and the one from the Spring manual is, that we provide a separate validator for each business rule instead of one per domain class. Of course many types of validators can easily be parameterised and re-used in different situations.

The business rules implementations must be invoked after every change of state to the object that could possibly violate the rule.