

# Zend\_Entity & Zend\_Db\_Mapper - Benjamin Eberlei

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

## Zend Framework: Zend\_Db\_Mapper Component Proposal

<b>Proposed Component Name</b>	Zend_Db_Mapper
<b>Developer Notes</b>	<a href="http://framework.zend.com/wiki/display/ZFDEV/Zend_Db_Mapper">http://framework.zend.com/wiki/display/ZFDEV/Zend_Db_Mapper</a>
<b>Proposers</b>	Benjamin Eberlei
<b>Zend Liaison</b>	TBD
<b>Revision</b>	1.0 - 25 January 2009: Initial Draft. 2.0 - 01 February 2009: Draft Ready for Recommendation (wiki revision: 41)

## Table of Contents

1. Overview
  - Discontinuing Zend Entity
  - Clarification of terms
  - Zend Package/Namespcae Discussion
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
- Scenarios (Integration Tests)
9. Class Skeletons

## 1. Overview

### Discontinuing Zend Entity

\*As a note for everyone finding this proposal: Zend Entity will be discontinued in favour of Doctrine 1 and 2 integration into ZF, see <http://www.mail-archive.com/fw-general@lists.zend.com/msg25412.html>\*

This component will be an implementation of **DataMapper pattern** that tries to separate Objects from their persistent separation to let you focus on the application more.

It will offer a huge range of features for handling all types of objects, versioning, relations, cascading operations. Extension points are provided to make the component as flexible as possible. The data mapping will be based on a mapping syntax that has to be written out explicitly (or be autogenerated). Data Mapping is a crucial point for developing applications that are developed **Domain Driven**. This component will contrast the Table-Row-Data-Gateway pattern which is more useful for Transaction Script or Table Module patterns of the model. Its **core feature** is the **focus on the Entity as datastructure** in contrast to SQL as datastructure. DataMapper makes heavy use of **LazyLoading for relations** that the actual related objects with Proxies that act as if they were the real objects and **only upon request** load the required data from the database. This lazyloading can be done for single related entities, collections or for fields of an entity that contain huge data (BLOBs and Text fields). The lazyloading replacements make sure that every relation of an object can be traversed/accessed inside the application without thinking about if it was loaded before.

Why is a data mapper a good extension to ZF? Because Data mapper is an enterprise pattern, and gets really useful in domain driven design. Zend Framework is an enterprise application framework. Other data mapper implementations in PHP are ezComponents Persistent Object (which leans to much to SQL Tables imho rather than on a Entity Class as the central datatype) and the under development FLOW3 and Doctrine 2.0 Frameworks.

The proposed data mapper solution will offer a generic solution and the possibility to extend certain parts of the mapper to match your needs perfectly. It will offer more flexibility for applications with complex business logic where a Table-Row-Data-Gateway will reach its limits faster. Using the DataMapper for an application as simple as a blog is nice, but rather overhead. This solution shines when you have more complex stuff going on.

This proposal is a first step, a second might be a proposal for **Zend\_Entity\_Repository** which implements the repository and specification patterns to completely encapsulate domain logic from persistence. Zend\_Repository might use adapter to offer access to Zend\_Db\_Table instances, Zend\_Db\_Mapper persistence sessions and even an adapter for In Memory Repositories which would significantly enhance Unit Testing of Domain Logic. It could also include adapters for another current proposal Zend\_CouchDb. Implementing the repository on top of a

functional mapper is peanuts though 😊

Additional further support to enhance this component would be integration with a Database Schema Component and different Tooling providers.

### Clarification of terms

**Entity** An entity is a class/object that encapsulates Domain Logic. It uses all object-oriented reference types like composition, aggregation and such to represent the state of business objects in memory. It has only little relation with the relational entity definition.

**Data Mapper** A class that maps entities into a relational database based on mapping definitions. It does that behind an API that hides the relational database. In short words: It maps the memory representation of objects into the database.

**Entity Manager** Single point of entry for calls to the underlying mapping persistence engine. It controls the entity definitions, unit-of-work, the mapping engine and it allows CRUD operations on those entities via load, save and delete methods. These methods accept SQL through the Zend\_Db\_Select Query object (only).

**Collections** Sets of related entities are saved in collections. These are ArrayAccess and Iterator implementations. Implementing those as objects rather than simple PHP arrays allows for Lazy Loading of collections behind the scenes and offer a way for the data mapper to recognize which related objects have changed, are new or were deleted in a session.

**Repository** A repository completely hides the underlying persistence from the business domain, it does not allow for sql or query objects. It offers access to objects via Criteria objects that specify which objects may be retrieved. An extended repository allows saving and deleting of objects. Criteria are transformed into the concrete selecting language, SQL in the case of relational databases.

**Loader** Based on the entity metadata definitions a loader implementation is selected for each entity to load the data from the database in the most efficient way.

**Persister** Inverted principle of the loader, based on the definition the persister knows how to persist an entity and its related entities.

### Zend Package/Namespace Discussion

Some of the interfaces and implementations of this proposal are that generic, that an ORM based on for example CouchDb could also use them. The overlap is fairly small though and in general I think the use-case of migrating from CouchDb to a RDMS or the other way around is rather small.

For this proposal to be really useful in a generic context it should follow its primary terminology Entity for its frontline interfaces. The question is whether separation into two namespaces "Zend\_Entity" and "Zend\_Db\_Mapper" are required, or if everything goes into subpackages of Zend\_Entity. Personally I would put it all under a new Zend\_Entity namespace although that might be misleading.

What is definitely required in my opinion is the definition of the following interfaces and subpackages in a new Zend\_Entity package that other persistence layers might be using (to name a few CouchDB, XML Databases, Remote Webservice that manage models).

- Zend\_Entity\_Manager\_Interface

- Zend\_Entity\_Interface
- Zend\_Entity\_Collection\_Interface
- Zend\_Entity\_IdentityMap
- Zend\_Entity\_Query\_\*
- Zend\_Entity\_MetadataFactory\_\*
- Zend\_Entity\_LazyLoad\_\*

All the stuff that currently resides inside "Zend\_Entity\_Mapper\_\*" is database specific.

## 2. References

- [Martin Fowler: Data Mapper Pattern](#)
- [Martin Fowler: Unit Of Work](#)
- [ezComponents Persistent Object](#)
- [FLOW 3 Persistence](#)
- [Doctrine](#)
- [Hibernate](#)
- [Eric Evans: Domain Driven Design](#)

## 3. Component Requirements, Constraints, and Acceptance Criteria

- Zend\_Db\_Mapper **is NOT** a model proposal. It merely allows to save Entities (classes) persistent. See "Domain Model, Table Module or Transaction Script" Patterns in Fowler PEAA for a model.
- Zend\_Db\_Mapper **will** put the Entity (A Record Class) as Datastructure into the focus, not the where and how it is saved for persistence.
  - An Entity is defined via a metadata syntax, by default a programmatic PHP implementation is shipped. Extension points for Annotations, XML or YAML configs are provided.
  - Metadata allows to specify: Property Names, Column Names, Property Types for example and many other details regarding the mapping.
  - Simple entity objects will be supported via an interface, which breaks encapsulation for PHP 5.2 objects
  - For PHP 5.3 usage a Reflection API will be implemented that directly sets properties (even private and protected ones).
- Zend\_Db\_Mapper **will** lean to a Data Mapper like Hibernate, it will provide the most basic functionality but offer rich interfaces to extend it.
  - It **will** take a Hibernate Mapping like syntax as basis to profit from the rich experience.
  - It will have reasonable defaults, only deviations should be configured.
  - It **will** provide a generic solution for the DataMapper pattern based on generic per entity data-mappers.
  - It **will** provide full functionality to define relations between tables that are translated into object composition behind the scenes.
  - Collections and object composition **will** make heavy use of behind the scenes lazy loading to encapsulate object creation and domain logic without hurting performance.
  - It **will** implement IdentityMap
  - It **will not** implement UnitOfWork so far. It can be added as a decorator to the Entity Manager Session with hooks into the event API at a later point.
  - It **will** provide a Query API for RDMS mappers that bases on Zend\_Db\_Select
  - It **will** provide a Query Object API that is sql-likeish but speaks in terms of the domain model (properties, entities).
- Zend\_Db\_Mapper **will** provide extension points through modularity that allow to overwrite specific functionality of the mapper.
  - It **will** strictly give responsibility of query building to subcomponents of the entity definitions such that developers can overwrite behaviour themselves (This compares to Doctrine behaviours)
  - Collection and lazy load classes **can** be configured to deviate from their defaults.
  - An event API **will** publish events on all entities.

## 4. Dependencies on Other Framework Components

- Zend\_Db\_Adapter\_Abstract
- Zend\_Db\_Select
- Zend\_Loader\_PluginLoader
- Zend\_Loader\_Autoloader\_Resource

## 5. Theory of Operation

### Steps of configuration:

- For each object that is an Entity you have to create a definition file that describes the process of mapping this entity to the database.
- Each object that is an Entity has to be implemented. An interface can be implemented that breaks up encapsulation of the entity for the datamapper.
- The Entity Manager will have to be initialized to be aware of the definitions.

### Steps of a session:

- A new **Entity Manager** is initializing with a Database connection and the mapping definitions.
- The **Entity Manager** is a single point of entry that encapsulates the **Mapping** and **Loading** of objects.
- The **Entity Manager** delegates all calls to the underlying concrete/generic mappers and the Query API.
- You can load Entities through the **Entity Manager** via Primary Key or a derived Zend\_Db\_Select object.
- The **Entity Manager** return and accept **Entity objects** that are controlled a very simple entity interface
- The **Entity Manager** delegates saving, selecting, updating and inserting.
- When **Entities** are retrieved the **Mapping** Engine decides how relations and large fields should be handled in terms of LazyLoading.
- **Entities** do not include any database related code (except hidden lazy loading mechanisms through an Inner/Outer Iterator schema).
- Besides more or less transactionless work, the session allows access to a Unit Of Work that handles a complete transaction based on the users need.

### Additional detail on my current concrete implementation proposal ideas:

- 1.) Concrete Type Mapper vs Generic Mapper: It will be allowed to overwrite the lowlevel mapper for a particular entity. By default all entities will use a powerful generic entity mapper, so that only when performance issues occur it should be relevant to write hard-coded sql by overwriting a mapper.
- 2.) The **Entity Manager** keeps track of all dependencies such as Db Adapter, the unit of work, identity map, entity definitions and the underlying mappers. They are enforced to be "singletons" with small "s" inside that session. This allows to have a hand on memory management which is necessary with UoW and Identity Map.
- 3.) LazyLoading will primarily be implemented at the Collection level, which under circumstances can lead to the N+1 query problem. This can be prevented by using outer joins which might be generated when loading the root object. You could also define a "formula" field can also be used to inject subselect values into an object field.
- 4.) Cascaded saving, updating and deleting of objects will be supported by configuration, allowing to save an entity and automatically saving all or some of its related entities.
- 5.) The generic implementation of the mapper is possible by using the Visitor pattern with the definition objects of a table. Depending on the state and action the mapper injects all the related data into a visitor accept function of the definition property. That way complicated lazy loading, association and other stuff is encapsulated at the point where its information is stored. This pattern allows for the great flexibility for developers to build their own properties with special handling that only need to be attached to the table definition and work without changes on the session and mapper code.
- 6.) As for every tool that tries to make you think less about the database this mapper implementation could cause considerable Database overhead, when used wrong. Therefore the documentation should by default include a section about pitfalls, performance and best-practices for special cases. To investigate the mapper behaviour I would like you to propose some object designs that we have to test in their persistent form.

## 6. Milestones / Tasks

- Milestone 1: Proposal (Done)
- Milestone 2: Early Prototype and add more Use-Cases from it (Done)
- Milestone 3: Reviews and Zend acceptance.
  - Milestone 3a: Integrate community feedback (lots of that is already done, more to come)
  - Milestone 3b: Integrate Zend feedback
- Milestone 4: Beta Phase with Iterative Feature Enhancements, Testing, Documentation and Collection of Scenarios
- Milestone 5: Release

## 7. Class Index

Why two namespaces? I discuss some requirements of that in the Component Overview section.

Core classes (Public API)

- `Zend_Entity_Manager_Interface`
- `Zend_Entity_IdentityMap`
- `Zend_Entity_Query_*`
- `Zend_Entity_MetadataFactory_*`
- `Zend_Entity_Manager_Interface`

Entity and Collection classes

- `Zend_Entity_Interface`
- `Zend_Entity_List` (Simple list 1...n of related objects)
- `Zend_Entity_Map` (Objects accessible via map-key => object association)
- `Zend_Entity_Set` (Unique objects)
- `Zend_Entity_Collection_Interface`
- `Zend_Entity_LazyLoad_*`

The namespace "`Zend_Db_Mapper_`" is currently "`Zend_Entity_Mapper_*`" in the svn and git repositories.

Database specific implementations of core interfaces

- `Zend_Db_Mapper_EntityManager`
- `Zend_Db_Mapper`
- `Zend_Db_Mapper_Select`
- `Zend_Db_Mapper_Query_*`

Loader and Persister classes

- `Zend_Db_Mapper_Persister_Interface`
- `Zend_Db_Mapper_Loader_Interface`
- A bunch of implementations based on the requirements of the entity definition

Definition classes (Tooling Providers would greatly enhance work with this)

- `Zend_Db_Mapper_DefinitionMap`
- `Zend_Db_Mapper_Definition_Entity` (**Core Definition**)
- `Zend_Db_Mapper_Definition_Property` (Simple column property mapped to entity field)
- `Zend_Db_Mapper_Definition_PrimaryKey` (Required Id field of entity)
- `Zend_Db_Mapper_Definition_CompositeKey`
- `Zend_Db_Mapper_Definition_Formula` (Read only properties that are SQL formulas, for example group formulas)
- `Zend_Db_Mapper_Definition_Collection` (Has Many collections of entities or value objects)
- `Zend_Db_Mapper_Definition_Join` (Join a second table that holds properties of entity)
- `Zend_Db_Mapper_Definition_Component` (Nested value object that is generated from table columns)
- `Zend_Db_Mapper_Definition_Discriminator` (Property that decides which sub-class to instantiate)
- `Zend_Db_Mapper_Definition_Version` (field to use for optimistic locking)
- `Zend_Db_Mapper_Definition_Timestamp` (Update timestamp on saving)
- `Zend_Db_Mapper_Definition_Date` (`Zend_Date` Serializer)
- `Zend_Db_Mapper_Definition_DateTime` (`DateTime` Serializer)
- `Zend_Db_Mapper_Definition_Id_Interface`
- `Zend_Db_Mapper_Definition_Id_AutoIncrement`
- `Zend_Db_Mapper_Definition_Id_Sequence`
- `Zend_Db_Mapper_Definition_Relation_Interface`
- `Zend_Db_Mapper_Definition_Relation_OneToMany`
- `Zend_Db_Mapper_Definition_Relation_ManyToOne`
- `Zend_Db_Mapper_Definition_Relation_OneToOne`
- `Zend_Db_Mapper_Definition_Relation_ManyToMany`

## 8. Use Cases

### Scenarios (Integration Tests)

I have implemented two scenarios for integration testing already, which are listed in the SVN repository:

<http://framework.zend.com/svn/framework/standard/branches/user/beberlei/ZendEntity/tests/Zend/Entity/IntegrationTest>

These show lots of use-cases and how the component works.

## 9. Class Skeletons

]]></ac:plain-text-body></ac:macro>  
]]></ac:plain-text-body></ac:macro>