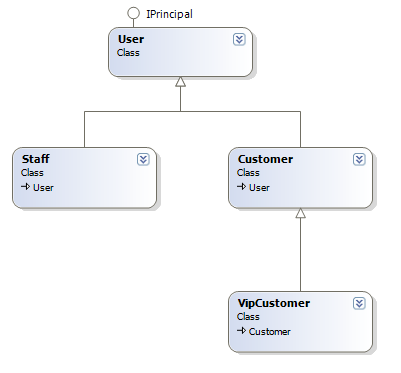
# Building Persistence Models with ActiveRecord

## Modelling Users

For users, we have to implement the following class tree:



The head of the sales department is an especially insisting culprit of a user and he talked our development leader in modeling VIP customers as a separate class. Later we can eventually change that, but for now we have to deal with that complicated tree.

Staff, on the other hand, is modeled more reasonable. We simply have a staff type and assign their duties and rights by adding roles to the members.

There are other, partly contradictory, requirements, as in every project. One manager just read about DDD and doesn’t want us to use a non-domain base class. Well, it’s just that he still demands that validation checking must be done automatically before saving. Our fellow developers are used to ActiveRecord and don’t want to cope with repositories or ActiveRecordMediator…

So, we’ll do our very best to satisfy them all…

## Test Project

We got principles, too. First we create a test project for simple smoke tests of our persistence model. First add all the references (they are in the code) and build a base class for in-memory-testing:

    1 public class TestBase : InMemoryTest

    2 {

    3     protected SessionScope scope;

    4

    5     [SetUp]

    6     public override void SetUp()

    7     {

    8         base.SetUp();

    9         scope = new SessionScope();

   10     }

   11

   12     [TearDown]

   13     public override void TearDown()

   14     {

   15         if (scope != null)

   16         {

   17             scope.Dispose();

   18             scope = null;

   19         }

   20         base.TearDown();

   21     }

   22

   23     public override Assembly[] GetAssemblies()

   24     {

   25         return new[] {typeof (User).Assembly};

   26     }

   27 }

Our test classes can now inherit this base class. ActiveRecord’s InMemoryTest base class doesn’t define any test-framework specific attributes, so you have to do this once for your favorite framework.

So we create a first test against a basic User class:

    1 [TestFixture]

    2 public class BasicUserTests : TestBase

    3 {

    4     [Test]

    5     public void ShouldBeAbleToCreateAnUser()

    6     {

    7         var user = new User()

    8                        {

    9                            Login = "bluechip1",

   10                            Email = "purchasing@bluechip.com",

   11                            Name = "John D. Farrow",

   12                            Password = "mommy"

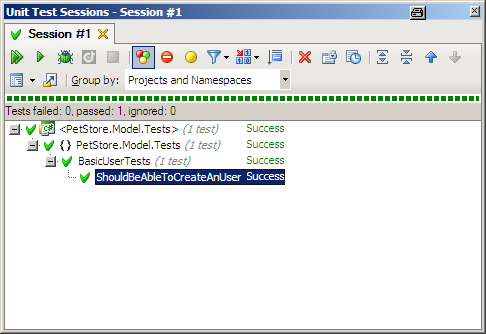
   13                        };

   14         ActiveRecordMediator<User>.Save(user);

   15     }

   16 }

With a basic User class (shown below), we get an early green result:



    1 [ActiveRecord("`User`",

    2     DiscriminatorColumn = "type",

    3     DiscriminatorType = "String",

    4     DiscriminatorValue = "user")]

    5 public class User : IPrincipal

    6 {

    7     [PrimaryKey(PrimaryKeyType.GuidComb)]

    8     public virtual Guid Id { get; protected set; }

    9

   10     [Property]

   11     public virtual string Login { get; set; }

   12

   13     [Property]

   14     public virtual string Name { get; set; }

   15

   16     [Property]

   17     public virtual string Email { get; set; }

   18

   19     [Property]

   20     public virtual string Password { get; set; }

   21

   22     public bool IsInRole(string role)

   23     {

   24         // We do not implement this functionality

   25         return false;

   26     }

   27

   28     public IIdentity Identity

   29     {

   30         get { return new GenericIdentity(Name, "castle.authentication"); }

   31     }

   32 }

The User class is quite simple with a few exceptions:

1. Since User is a reserved name in most databases, we set a table name and escape it with back ticks, because User is a fine name for the table.
2. We’re using Discriminator attributes in lines 2-4 because we are going to set up a class tree with the table-per-hierarchy pattern.
3. All properties are marked as virtual, so that we can use lazy loading. This is a best practice: Don’t be fooled into omitting it as YAGNI; lazy loading is such a common requirement that you should plan for it from the beginning.

## Validation

Someone said something about validation, so we need to add some basic validations to the user class. For now, let’s say that all properties except Id are mandatory and that Login must be unique. We can do these validations using two different ways:

### Database Constraints

We’re adding NotNull and Unique attributes to the properties. When we are CreateSchema() the table columns will have the desired constraints.

When using this approach, both the database and NHibernate check that the value is not null and that it is unique respectively.

Unfortunately, it has a major drawback: We cannot check in advance whether an entity can be saved to the database without an error. If we tried and encountered an error, we had to dump and recreate the NHibernate session. We would lose all information about other dirty entities in that case.

Also, we can only check a small number of constraints this way. For example, we cannot check if a valid email address is given or just an empty string.

### Validation framework

In addition to constraints, we can use a validation framework that allows us to specify our validity rules in the model instead of the database. We can then check that our data is valid before we send it to the database.

As a matter of coincidence, Castle ships with its own validation framework, so we are going to use that here.

### Implementing validation

To implement validation on our user type, we will use Castle Validator. We can use database constraints in addition, as a last line of defense against invalid data, but the main validation work will be done in the model, not in the database.

First, we create a short, failing test:

    1 [Test]

    2 public void CannotCreateAnUserWithoutEmail()

    3 {

    4     Assert.Throws<ValidationException>(() =>

    5        ActiveRecordMediator<User>.Save(

    6            new User {Login = "none", Name = "A. N. Onymous", Password = ""}

    7            ));

    8 }

To make it pass, we add the required attribute to the Email-property of the user class:

[Property(NotNull = true)]

[ValidateNonEmpty]

public virtual string Email { get; set; }

We leave the other properties as they are for now to verify our approach. We can still add validation to them afterwards.

But, the test is still failing. We get an exception, but not the desired one: NHibernate refuses to save the entity. The last line of defense is still standing, but the session is flawed.

The reason is that the validation framework itself is never called. It would have been called if we used ActiveRecordValidationBase as a base class, but we have POCOs, so there is no automatic validation available. Let’s do it manually.

We now change the test, so that NHibernate is satisfied and we can verify that the validation framework refuses to save the entity:

    1 [Test]

    2 public void CannotCreateAnUserWithoutEmail()

    3 {

    4     Assert.Throws<ValidationException>(() =>

    5        ActiveRecordMediator<User>.Save(

    6            new User {

    7             Login = "", Name = "A. N. Onymous",

    8             Password = "",Email=""}

    9            ));

   10 }

First, we ignore the failing test and create a new one that checks that by manual validation, we can find the missing email in advance:

    1 [Test]

    2 public void AnUserWithoutEmailIsInvalid()

    3 {

    4     IValidatorRegistry registry = new CachedValidationRegistry();

    5     IValidatorRunner runner = new ValidatorRunner(registry);

    6     var user = new User

    7                    {

    8                        Login = "", Name = "A. N. Onymous",

    9                     Password = "", Email = ""

   10                    };

   11     Assert.That(runner.IsValid(user), Is.False);

   12 }

This test passes on the first try. But we still want the test before to pass: Invalid entities must never make it to the database.

To ensure such an orthogonal behavior, we can make use of NHibernate’s event listeners. ActiveRecord allows us to add them to the model quite simply:

    1 [EventListener]

    2 public class ValidationChecker :

    3     IPreUpdateEventListener, IPreInsertEventListener

    4 {

    5     public bool OnPreUpdate(PreUpdateEvent @event)

    6     {

    7         ThrowIfInvalid(@event.Entity);

    8         return false;

    9     }

   10

   11     public bool OnPreInsert(PreInsertEvent @event)

   12     {

   13         ThrowIfInvalid(@event.Entity);

   14         return false;

   15     }

   16

   17     private static IValidatorRegistry registry =

   18         new CachedValidationRegistry();

   19     private IValidatorRunner runner =

   20         new ValidatorRunner(registry);

   21

   22     private void ThrowIfInvalid(object o)

   23     {

   24         if (!runner.IsValid(o))

   25         {

   26             throw new ValidationException(

   27                 string.Format(

   28                 "Validation of {0} has failed", o));

   29         }

   30     }

   31 }

The listener is registered with NHibernate by using the EventListener-attribute in line 1. It will be registered for all the Listener-interfaces it implements. We therefore need only one listener for both insert and update commands. The work is done in lines 22-30. This could and should be done better. For example, one could not throw an exception, but simply return “true” to veto the change and raise a message so that the problem could be handled gracefully in higher level code.

## Data Access

We have now only one thing missing: How should we access our entities?

The default option would be using ActiveRecordBase, but we cannot use that here. The basic alternative is using ActiveRecordMediator, but this has some drawbacks:

1. As we have seen, there is no validation executed when an entity is saved.
2. There is no possibility to use doubles for unit testing.
3. ActiveRecordMediator<MyEntity>.Save(myEntity) is quite long to type or read.

Can we bring all of these requirements together? Ideally, can we somehow emulate ActiveRecord-like syntax sugar like entity.Save()? The rest of the chapter deals with doing exactly this.

Another option we have is using a Data Access Object (DAO), which is nowadays often found relabeled as Repository. DAOs allow using test doubles and we can make them call our validation before we do the actual data manipulation. Unfortunately, we had to manage our DAOs then, but at least it’s a start.

We therefore define an interface for our DAOs:

    1 public interface IDao<TClass> where TClass: class, IAggregateRoot

    2 {

    3     void Create(TClass t);

    4     void Update(TClass t);

    5     void Save(TClass t);

    6     void Delete(TClass t);

    7

    8     TClass Find(object id);

    9     IQueryable<TClass> Linq();

   10     ICriteria GetCriteria();

   11     ICriteria GetCriteria(string alias);

   12     IQuery GetQuery(string query);

   13     IQuery GetNamedQuery(string name);

   14 }

The interface contains definitions for basic data access operations. This interface is expected to grow along with its implementers when other operations are needed, so it is the best to define a default implementation that can be extended instead of implementing the interface.

    1 public class GenericDao<TClass> :IDao<TClass> where TClass:class, IAggregateRoot

    2 {

    3     private IValidatorRegistry registry;

    4     private IValidatorRunner runner;

    5

    6     public virtual IValidatorRegistry Registry

    7     {

    8         get { return registry; }

    9         set { registry = value; }

   10     }

   11

   12     public virtual IValidatorRunner Runner

   13     {

   14         get { return runner; }

   15         set { runner = value; }

   16     }

   17

   18     public GenericDao()

   19     {

   20         registry = new CachedValidationRegistry();

   21         runner = new ValidatorRunner(registry);

   22     }

   23

   24     public void Create(TClass t)

   25     {

   26         Validate(t);

   27         ActiveRecordMediator<TClass>.Create(t);

   28     }

   29

   30     protected virtual void Validate(TClass t)

   31     {

   32         if (!Runner.IsValid(t))

   33         {

   34             var errors = Runner.GetErrorSummary(t);

   35             throw new ValidationException(string.Format("{0} is not valid", typeof (TClass).FullName), errors.ErrorMessages);

   36         }

   37     }

The above source code is not complete; it omits most interface implementations for brevity, showing just the idea behind the class: Do whatever additional work is necessary and delegate the actual data access to ActiveRecordMediator.

I have not explained a certain issue of the previous code samples: Both have a class constraint that allows only implementers of IAggregateRoot to be used as a type parameter. IAggregateRoot is an empty marker interface, by the way.

From Domain Driven Design, I especially appreciate the idea of Bounded Contexts and Aggregates. The aggregate concept defines responsibility boundaries for persistence. So if I have a user and some types that depend on it, for example account data for customers, I don’t want anyone to access the account data directly, but only by using a customer. Customer is therefore the aggregate root and the Account class is part of the Customer’s aggregate.

By constraining the type parameter to be of type IAggregateRoot, I do not only prevent nonsense DAOs like GenericDao<string>, but also enforce that only defined aggregate roots can be explicitly persisted. Dependent objects must be persisted by reachability, so we have to define proper cascading for them.

Now that we have a DAO, we want to enable our fellow programmers to use it as conveniently as possible. So I define a class with static methods so that it is not necessary to manage the DAO manually anymore.

    1 public class Storage<T> where T:class, IAggregateRoot

    2 {

    3     private static IDao<T> dao = new GenericDao<T>();

    4

    5     public static void RegisterDao(IDao<T> replacement)

    6     {

    7         dao = replacement;

    8     }

    9

   10     public static void Create(T t)

   11     {

   12         dao.Create(t);

   13     }

Again, most methods have been left out for brevity. The code shows the concept using the Create-method. It also shows how we can replace the actual DAO with a test double: by calling RegisterDao with a mock object.

We are now one step short of emulating the original ActiveRecord-syntax. What’s left is defining extension methods for IAggregateRoot to enable calling operations directly from objects:

    1 public static class StorageExtensions

    2 {

    3     public static void Save<T>(this T t) where T:class, IAggregateRoot

    4     {

    5         Storage<T>.Save(t);

    6     }

This now allows our code to pass the following test:

    1 [Test]

    2 public void SupportsActiveRecordSemantics()

    3 {

    4     var user = new User()

    5     {

    6         Login = "bluechip1",

    7         Email = "purchasing@bluechip.com",

    8         Name = "John D. Farrow",

    9         Password = "mommy"

   10     };

   11

   12     user.Save();

   13

   14     var users = (from u in Storage<User>.Linq()

   15                  where u.Email.Contains("blue")

   16                  select u).ToList();

   17     Assert.That(users.Count,Is.EqualTo(1));

   18 }

Additionally, we can create tests by mocking data access, so that we don’t even have to set up ActiveRecord for testing:

    1 [TestFixture]

    2 public class MockingTest

    3 {

    4     [Test]

    5     public void CanMockUserAccess()

    6     {

    7         var dao = MockRepository.GenerateMock<IDao<User>>();

    8         dao

    9             .Expect(d => d.Save(null))

   10             .IgnoreArguments()

   11             .Repeat.AtLeastOnce();

   12         Storage<User>.RegisterDao(dao);

   13

   14         (new User()).Save();

   15

   16         dao.VerifyAllExpectations();

   17     }

   18 }

That way, we can use the ActiveRecord-syntax and still mock data access. The above test needs only a quarter of the time to run an in-memory-test with ActiveRecord initialized. When we are using a real database for testing, we would need even longer.

## What was achieved?

By now, a lot of requirements had been fulfilled:

* No framework-specific base class has been used.
* Validity of the data is ensured by inspecting all changes before they are sent to the database.
* We have a self-servicing ActiveRecord-like syntax that
  + is restricted to aggregate roots
  + frees us from managing DAOs
  + and allows to mock data access