
A hands-on introduction to Wt::Dbo

Table of Contents

1. Introduction	1
2. Mapping a single class	2
3. A first session	3
4. Querying objects	5
5. Updating objects	7
6. Mapping relations	8
6.1. <i>Many-to-One</i> relations	8
6.2. <i>Many-to-Many</i> relations	10
6.3. <i>One-to-One</i> relations	12
7. Customizing the mapping	12
7.1. Changing or disabling the surrogate primary key "id" field	12
7.2. Changing or disabling the "version" field	13
7.3. Specifying a natural primary key	13
7.4. Specifying a composite natural primary key	14
7.5. Specifying foreign key constraints	15
7.6. Specifying a natural primary key that is also a foreign key	16
8. Transactions and concurrency	18
9. Installation	19

Koen Deforche <koen@emweb.be>

For Wt 3.2.0 (July 14 2011)

1. Introduction

Wt::Dbo is a C++ ORM (Object-Relational-Mapping) library.

The library is distributed as part of Wt [<http://www.webtoolkit.eu/wt>] for building database-driven web applications, but may be equally well used independently from it.

The library provides a class-based view on database tables which keeps an object-hierarchy of database object automatically synchronized with a database by inserting, updating and deleting database records. C++ classes map to database tables, class fields to table columns, and pointers and collections of pointers to database relations. An object from a mapped class is called a **database object** (dbo). Query results may be defined in terms of database objects, primitives, or tuples of these.

A modern C++ approach is used to solve the mapping problem. Rather than resorting to XML-based descriptions of how C++ classes and fields should map onto tables and columns, or using obscure macros, the mapping is defined entirely in C++ code.

In this tutorial, we will work our way through a blogging example, similar to the one that is distributed with the library.

Tip

The complete source code for the examples used in this tutorial are available as ready-to-run programs in the `examples/feature/dbo/` folder of Wt [<http://www.webtoolkit.eu/wt/download>].

2. Mapping a single class

We will start off with using `Wt::Dbo` for mapping a single class `User` to a corresponding table `user`.

Warning

In this tutorial and the examples, we alias the namespace `Wt::Dbo` to `dbo`, and in our explanation we will refer to types and methods available in that namespace directly.

To build the following example, you need to link against the `wtdbo` and `wtdbosqlite3` libraries.

Mapping a single class (tutorial1.C).

```
#include <Wt/Dbo/Dbo>
#include <string>

namespace dbo = Wt::Dbo;

class User {
public:
    enum Role {
        Visitor = 0,
        Admin = 1,
        Alien = 42
    };

    std::string name;
    std::string password;
    Role        role;
    int         karma;

    template<class Action>
    void persist(Action& a)
    {
        dbo::field(a, name,      "name");
        dbo::field(a, password, "password");
        dbo::field(a, role,     "role");
        dbo::field(a, karma,   "karma");
    }
};
```

This example shows how persistence support is defined for a C++ class. A template member method `persist()` is defined which serves as a persistence definition for the class. For each member in the class, a call to `Wt::Dbo::field()` is used to map the field to a table column name.

As you may see, standard C++ types such as `int`, `std::string` and `enum` types are readily supported by the library (a full list of supported types can be found in the documentation of `Wt::Dbo::sql_value_traits<T>`). Support for other types can be added by specializing

`Wt::Dbo::sql_value_traits<T>`. There is also support for built-in Wt types such as `WDate`, `WDateTime`, `WTime` and `WString` which can be enabled by including `<Wt/Dbo/WtSqlTraits>`.

The library defines a number of actions which will be applied to a database object using its `persist()` method, which applies it in turn to all its members. These actions will then read, update or insert database objects, create the schema, or propagate transaction outcomes.

Note

For brevity, our example uses public members. There is nothing that prevents you to encapsulate your state in private members and provide accessor methods. You may even define the persistence method in terms of accessor methods by differentiating between read and write actions.

3. A first session

Now that we have a mapping definition for our `User` class, we can start a database session, create our schema (if necessary) and add a user to the database.

Let us walk through the code for doing this.

(tutorial1.C continued).

```
void run()
{
    /*
     * Setup a session, would typically be done once at application startup.
     */
    dbo::backend::Sqlite3 sqlite3("blog.db");
    dbo::Session session;
    session.setConnection(sqlite3);

    ...
}
```

The `Session` object is a long living object that provides access to our database objects. You will typically create a `Session` object for the entire lifetime of an application session, and one per user. None of the `Wt::Dbo` classes are thread safe (except for the connection pools), and session objects are not shared between sessions.

The lack of thread-safety is not simply a consequence of laziness on our part. It coincides with the promises made by transactional integrity on the database: you will not want to see the changes made by one session in another session while its transaction has not been committed (Read-Committed transaction isolation level). It might make sense however to implement a copy-on-write strategy in the future, to allow sharing of the bulk of database objects between sessions.

The session is given a connection which it may use to communicate with the database. A session will use a connection only during a transaction, and thus does not really need a dedicated connection. When you are planning for multiple concurrent sessions, it makes sense to use a connection pool instead, and a session may also be initialized with a reference to a connection pool.

`Wt::Dbo` uses an abstraction layer for database access, and currently supports Postgres and Sqlite3 as backends.

(tutorial1.C continued).

```
...  
  
session.mapClass<User>( "user" );  
  
/*  
 * Try to create the schema (will fail if already exists).  
 */  
session.createTables();  
  
...
```

Next, we use `mapClass()` to register each database class with the session, indicating the database table onto which the class must be mapped.

Certainly during development, but also for initial deployment, it is convenient to let `Wt::Dbo` create or drop the database schema.

This generates the following SQL:

```
begin transaction  
create table "user" (  
    "id" integer primary key autoincrement,  
    "version" integer not null,  
    "name" text not null,  
    "password" text not null,  
    "role" integer not null,  
    "karma" integer not null  
)  
commit transaction
```

As you can see, next to the four columns that map to C++ fields, by default, `Wt::Dbo` adds another two columns: `id` and `version`. The `id` is a surrogate primary key, and `version` is used for version-based optimistic locking. Since Wt 3.1.4, `Wt::Dbo` you can suppress the version field, and provide natural keys of any type instead of the surrogate primary key, see [Customizing the mapping](#).

Finally, we can add a user to the database. All database operations happen within a transaction.

(tutorial1.C continued).

```
...  
/*  
 * A unit of work happens always within a transaction.  
 */  
dbo::Transaction transaction(session);  
  
User *user = new User();  
user->name = "Joe";  
user->password = "Secret";  
user->role = User::Visitor;  
user->karma = 13;  
  
dbo::ptr<User> userPtr = session.add(user);  
  
transaction.commit();  
}
```

A call to `Session::add()` adds an object to the database. This call returns a `ptr<Dbo>` to reference a database object of type `Dbo`. This is a shared pointer which also keeps track of the persistence state of the referenced object. Within each session, a database object will be loaded at most once: the session keeps track of loaded database objects and returns an existing object whenever a query to the database requires this. When the last pointer to a database object goes out of scope, the *transient* (in-memory) copy of the database object is also deleted (unless it was modified, in which case the transient copy will only be deleted after changes have been successfully committed to the database).

The session also keeps track of objects that have been modified and which need to be flushed (using SQL statements) to the database. Flushing happens automatically when committing the transaction, or whenever needed to maintain consistency between the transient objects and the database copy (e.g. before doing a query).

This generates the following SQL:

```
begin transaction
insert into "user" ("version", "name", "password", "role", "karma")
    values (?, ?, ?, ?, ?)
commit transaction
```

All SQL statements are prepared once (per connection) and reused later, which has the benefit of avoiding SQL injection problems, and allows potentially better performance.

4. Querying objects

There are two ways of querying the database. Database objects of a single `Dbo` class can be queried using `Session::find<Dbo>(condition)`:

(tutorial1.C continued).

```
dbo::ptr<User> joe = session.find<User>().where("name = ?").bind("Joe");
std::cerr << "Joe has karma: " << joe->karma << std::endl;
```

All queries use prepared statements with positional argument binding. The `Session::find<T>()` method returns a `Query< ptr<T> >` object. The `Query` object can be used to refine the search by defining `Sql` `where`, `order by` and `group by` definitions, and allows binding of parameters using `Query::bind()`. In this case the query should expect a single result and is casted directly to a database object pointer.

Note

Since Wt 3.1.3, the `Query` class has a second parameter `BindStrategy` which has two possible values, corresponding two different query implementations.

The default strategy is *DynamicBinding* and allows the query to be a long lived object associated with the session which may be run multiple times. It also allows you to modify the query by changing only the order or the limit/offsets.

An alternative strategy is *DirectBinding* which passes bound parameters directly on to an underlying prepared statement. This corresponds to the old behavior of a `Query` object.

Such a query can be run only once, but has the benefit of having less (C++) overhead because the parameters values are directly passed on to the backend instead of stored within the query object.

The query formulated to the database is:

```
select id, version, "name", "password", "role", "karma"
  from "user"
 where (name = ?)
```

The more general way for querying uses `Session::query<Result>(sql)`, which supports not only database objects as results. The query of above is equivalent to:

(tutorial1.C continued).

```
dbo::ptr<User> joe2 = session.query< dbo::ptr<User> >("select u from user u")
  .where("name = ?").bind("Joe");
```

And this generates similar SQL:

```
select u.id, u.version, u."name", u."password", u."role", u."karma"
  from user u
 where (name = ?)
```

The `sql` statement passed to the method may be arbitrary `sql` which returns results that are compatible with the `Result` type. The `select` part of the SQL query may be rewritten (as in the example above) to return the individual fields of a queried database object.

To illustrate that `Session::query<Result>()` may be used to return other types, consider the query below where an `int` result is returned.

(tutorial1.C continued).

```
int count = session.query<int>("select count(1) from user")
  .where("name = ?").bind("Joe");
```

The queries above were expecting unique results, but queries can also have multiple results. A `Session::query<Result>()` may therefore in general return a `dbo::collection< Result >` (for multiple results) and in the examples above they were coerced to a single unique `Result` for convenience. Similarly, `Session::find<Dbo>()` may return a `collection< ptr<Dbo> >` or a unique `_ptr<Dbo>`. If a unique result is asked, but the query found multiple results, a `NoUniqueResultException` will be thrown.

`collection<T>` is an STL-compatible collection which has iterators that implement the `InputIterator` requirements. Thus, you can only iterate through the results of a collection once. After the results have been iterated the `collection` can no longer be used (but the `Query` object can be reused unless a `DirectBinding` bind strategy was used).

The following code shows how you may multiple results of a query may be iterated:

(tutorial1.C continued).

```
typedef dbo::collection< dbo::ptr<User> > Users;
Users users = session.find<User>();
```

```
std::cerr << "We have " << users.size() << " users:" << std::endl;

for (Users::const_iterator i = users.begin(); i != users.end(); ++i)
    std::cerr << " user " << (*i)->name
                << " with karma of " << (*i)->karma << std::endl;
```

This code will perform two database queries: one for the call to `collection::size()` and one for iterating the results:

```
select count(1) from "user"
select id, version, "name", "password", "role", "karma" from "user"
```

Warning

A query uses a prepared statement to execute, and prepares a new statement if no statement was yet prepared for that query. Because a prepared statement is usually not reentrant and at the same time a query will use an existing statement if one exists, you need to be careful to not have two collections with the same statement busy at the same time. Thus while iterating the results of a query you cannot use that same query again. Therefore it may be necessary to copy the results into a standard container (such as `std::vector`) before iterating them. Since Wt 3.1.3, concurrent use will be detected and an exception will be thrown saying:

```
A collection for '...' is already in use. Reentrant statement use is
not yet implemented.
```

We plan to remove this restriction in later versions, cloning prepared statements as necessary.

5. Updating objects

Unlike most other smart pointers, `ptr<Dbo>` is read-only by default: it returns a `const Dbo*`. To modify a database object, you need to call the `ptr::modify()` method, which returns a non-const object. This mark the object as dirty and the modifications will later be synchronized to the database.

(tutorial1.C continued).

```
dbo::ptr<User> joe = session.find<User>().where("name = ?").bind("Joe");

joe.modify()->karma++;
joe.modify()->password = "public";
```

Database synchronization does not happen instantaneously, instead, they are delayed until explicitly asked, using `ptr<Dbo>::flush()` or `Session::flush()`, until a query is executed whose results may be affected by the changes made, or until the transaction is committed.

The previous code will generate the following SQL:

```
select id, version, "name", "password", "role", "karma"
  from "user"
  where (name = ?)
update "user"
```

```
set "version" = ?, "name" = ?, "password" = ?, "role" = ?, "karma" = ?  
where "id" = ? and "version" = ?
```

We already saw how using `Session::add(ptr<Dbo>)`, we added a new object to the database. The opposite operation is `ptr<Dbo>::remove()`: it deletes the object in the database.

(tutorial1.C continued).

```
dbo::ptr<User> joe = session.find<User>().where("name = ?").bind("Joe");  
joe.remove();
```

After removing an object, the transient object can still be used, and can even be re-added to the database.

Note

Like `modify()`, also the `add()` and `remove()` operations defer synchronization with the database, and therefore the following code does not actually have any effect on the database:

(tutorial1.C continued).

```
dbo::ptr<User> silly = session.add(new User());  
silly.modify()->name = "Silly";  
silly.remove();
```

6. Mapping relations

6.1. *Many-to-One* relations

Let's add posts to our blogging example, and define a Man-to-One relation between posts and users. In the code below, we limit ourselves to the statements important for defining the relationship.

Many-to-One relation (tutorial2.C).

```
#include <Wt/Dbo/Dbo>  
#include <string>  
  
namespace dbo = Wt::Dbo;  
  
class User;  
  
class Post {  
public:  
    ...  
  
    dbo::ptr<User> user;  
  
    template<class Action>  
    void persist(Action& a)  
    {  
        ...  
    }  
};
```

```
        dbo::belongsTo(a, user, "user");
    }
};

class User {
public:
    ...

    dbo::collection< dbo::ptr<Post> > posts;

    template<class Action>
    void persist(Action& a)
    {
        ...

        dbo::hasMany(a, posts, dbo::ManyToOne, "user");
    }
};
```

At the *Many*-side, we add a reference to a user, and in the `persist()` method we call `belongsTo()`. This allows us to reference the user to which this post belongs. The last argument will correspond to the name of the database column which defines the relationship.

At the *One*-side, we add a collection of posts, and in the `persist()` method we call `hasMany()`. The join field must be the same name as in reciprocal `belongsTo()` method call.

If we add the `Post` class too to our session using `Session::mapClass()`, and create the schema, the following SQL is generated:

```
create table "user" (
    ...

    -- table user is unaffected by the relationship
);

create table "post" (
    ...

    "user_id" bigint,
    constraint "fk_post_user" foreign key ("user_id") references "user" ("id")
)
```

Note the `user_id` field which corresponds to the join name “user”.

At the *Many*-side, you may read or write the `ptr` to set a user to which this post belongs.

The collection at the *One*-side allows us to retrieve all associated elements, and also `insert()` and `remove()` elements, which has the same effect as setting the `ptr` on the *Many*-side.

Example:

(tutorial2.C continued).

```
dbo::ptr<Post> post = session.add(new Post());
post.modify()->user = joe; // or joe.modify()->posts.insert(post);

// will print 'Joe has 1 post(s).'
```

```
std::cerr << "Joe has " << joe->posts.size() << " post(s)." << std::endl;
```

As you can see, as soon as *joe* is set as *user* for the new post, the post is reflected in the *posts* collection of *joe*, and vice-versa.

Warning

The collection uses a prepared statement to execute. Collections will try to share a single prepared statement, but prepared statements are usually not reentrant. As a result, you need to be careful to not have two collections with the same statement busy at the same time. Thus while iterating a collection, you need to be sure you will not reentrantly iterate the same collection (of the same or another object). Therefore it may be necessary to copy the results into a standard container (such as `std::vector`) before iterating them.

We plan to remove this restriction in later versions, cloning prepared statements as necessary.

6.2. *Many-to-Many* relations

To illustrate *Many-to-Many* relations, we will add tags to our blogging example, and define an *Many-to-Many* relation between posts and tags. In the code below, we again limit ourselves to the statements important for defining the relationship.

Many-to-Many relation (tutorial2.C).

```
#include <Wt/Dbo/Dbo>
#include <string>

namespace dbo = Wt::Dbo;

class Tag;

class Post {
public:
    ...

    dbo::collection< dbo::ptr<Tag> > tags;

    template<class Action>
    void persist(Action& a)
    {
        ...

        dbo::hasMany(a, tags, dbo::ManyToMany, "post_tags");
    }
};

class Tag {
public:
    ...

    dbo::collection< dbo::ptr<Post> > posts;

    template<class Action>
    void persist(Action& a)
    {
```

```
...
    dbo::hasMany(a, posts, dbo::ManyToMany, "post_tags");
}
};
```

As expected, the relationship is reflected in almost the same way in both classes: they both have a collection of database objects of the related class, and in the `persist()` method we call `hasMany()`. The join field in this case will correspond to the name of a join-table used to persist the relation.

Adding the `Post` class to our session using `Session::mapClass()`, we now get the following SQL for creating the schema:

```
create table "post" (
    ...
    -- table post is unaffected by the relationship
)

create table "tag" (
    ...
    -- table tag is unaffected by the relationship
)

create table "post_tags" (
    "post_id" bigint not null,
    "tag_id" bigint not null,
    primary key ("post_id", "tag_id"),
    constraint "fk_post_tags_key1" foreign key ("post_id")
        references "post" ("id"),
    constraint "fk_post_tags_key2" foreign key ("tag_id")
        references "tag" ("id")
)

create index "post_tags_post" on "post_tags" ("post_id")
create index "post_tags_tag" on "post_tags" ("tag_id")
```

The collection at either side of the *Many-to-Many* relation allows us to retrieve all associated elements. Unlike a collection in a *Many-to-One* relation however, we may now also `insert()` and `erase()` items from the collection. To define a relation between a post and a tag, you need to add the post to the tag's *posts* collection, or the tag to the post's *tags* collection. You may not do both! The change will automatically be reflected in the reciprocal collection. Likewise, to undo the relation between a post and a tag, you should remove the tag from the post's *tags* collection, or the post from the tag's *posts* collection, but not both.

Example:

(tutorial2.C continued).

```
dbo::ptr<Post> post = ...
dbo::ptr<Tag> cooking = session.add(new Tag());
cooking.modify()->name = "Cooking";

post.modify()->tags.insert(cooking);
```

```
// will print '1 post(s) tagged with Cooking.'  
std::cerr << cooking->posts.size() << " post(s) tagged with Cooking."  
    << std::endl;
```

Warning

The same warning as above applies here as well.

6.3. One-to-One relations

One-to-One relations are currently not supported, but can be simulated using *Many-to-One* relations as they have the same database schema structure.

7. Customizing the mapping

By default, `Wt::Dbo` will add an auto-incrementing surrogate primary (`id`) key and a version field (`version`) to each mapped table.

While these defaults make sense for a new project, you can tailor the mapping so that you can map to virtually any existing database schema.

7.1. Changing or disabling the surrogate primary key "id" field

To change the field name used for the surrogate primary key for a mapped class, or to disable the surrogate primary key for a class and use a neutral key instead, you need to specialize `Wt::Dbo::dbo_traits<C>`.

For example, the code below changes the primary key field for class `Post` from `id` to `post_id`:

Changing the "id" field name (tutorial3.C).

```
#include <Wt/Dbo/Dbo>  
  
namespace dbo = Wt::Dbo;  
  
class Post {  
public:  
    ...  
};  
  
namespace Wt {  
    namespace Dbo {  
  
        template<>  
        struct dbo_traits<Post> : public dbo_default_traits {  
            static const char *surrogateIdField() {  
                return "post_id";  
            }  
        };  
  
    }  
}
```

7.2. Changing or disabling the "version" field

To change the field name used for the optimistic concurrency control version field (`version`), or to disable optimistic concurrency control for a class altogether, you need to specialize `Wt::Dbo::dbo_traits<C>`.

For example, the code below disables optimistic concurrency control for class `Post`:

Disabling the "version" field name (tutorial4.C).

```
#include <Wt/Dbo/Dbo>

namespace dbo = Wt::Dbo;

class Post {
public:
    ...
};

namespace Wt {
    namespace Dbo {

        template<>
        struct dbo_traits<Post> : public dbo_default_traits {
            static const char *versionField() {
                return 0;
            }
        };

    }
}
```

7.3. Specifying a natural primary key

Instead of using a auto-incrementing surrogate primary key, you may want to use a different primary key.

For example, the code below changes the primary key for the `User` table to a string (his username) which maps onto a `varchar (20)` field `user_name`:

Using a natural key (tutorial5.C).

```
#include <Wt/Dbo/Dbo>

namespace dbo = Wt::Dbo;

class User {
public:
    std::string userId;

    template<class Action>
    void persist(Action& a)
    {
        dbo::id(a, userId, "user_id", 20);
    }
};
```

```
namespace Wt {
    namespace Dbo {

        template<>
        struct dbo_traits<User> : public dbo_default_traits {
            typedef std::string IdType;

            static IdType invalidId() {
                return std::string();
            }

            static const char *surrogateIdField() { return 0; }
        };
    }
}
```

The `id()` function has the same syntax as the `field()` function.

A natural primary key may also be a composite key, a foreign key or a combination.

7.4. Specifying a composite natural primary key

To use a composite type as a natural primary key, i.e. a type which consists of more than one field, you need to have a corresponding C++ type.

The type has a number of basic requirements, such as default constructor, comparison operators (`==` and `<`), and a streaming operator.

Using a composite natural primary key (tutorial6.C).

```
struct Coordinate {
    int x, y;

    Coordinate()
        : x(-1), y(-1) { }

    Coordinate(int an_x, int an_y)
        : x(an_x), y(an_y) { }

    bool operator== (const Coordinate& other) const {
        return x == other.x && y == other.y;
    }

    bool operator< (const Coordinate& other) const {
        if (x < other.x)
            return true;
        else if (x == other.x)
            return y < other.y;
        else
            return false;
    }
};

std::ostream& operator<< (std::ostream& o, const Coordinate& c)
{
    return o << "(" << c.x << ", " << c.y << ")";
}
```

Next, you must indicate how the type is persisted, by overloading Dbo's `field()` function for it.

(tutorial6.C continued).

```
namespace Wt {
    namespace Dbo {

        template <class Action>
        void field(Action& action, Coordinate& coordinate,
                  const std::string& name, int size = -1)
        {
            field(action, coordinate.x, name + "_x");
            field(action, coordinate.y, name + "_y");
        }
    }
}
```

With this in place, we can use the `Coordinate` type as a natural primary key type:

(tutorial6.C continued).

```
class GeoTag;

namespace Wt {
    namespace Dbo {

        template<>
        struct dbo_traits<GeoTag> : public dbo_default_traits
        {
            typedef Coordinate IdType;
            static IdType invalidId() { return Coordinate(); }
            static const char *surrogateIdField() { return 0; }
        };
    }
}

class GeoTag {
public:
    Coordinate position;
    std::string name;

    template <class Action>
    void persist(Action& a)
    {
        dbo::id(a, position, "position");
        dbo::field(a, name, "name");
    }
};
```

Note that the composite key may also include foreign keys, by storing `ptr<>` objects in the composite, which you map using a `belongsTo()` declaration. See `tutorial8.C` for a complete example.

7.5. Specifying foreign key constraints

The `belongsTo()` function is overloaded so that you can add foreign key constraints which are enforced by the database, such as:

- `NotNull`: cannot be null
- `OnUpdateCascade`: cascade an update of the (natural) primary key to the foreign keys that reference it
- `OnUpdateSetNull`: an update of the (natural) primary key sets referencing foreign keys to null
- `onDeleteCascade`: cascade a delete of the object to also delete objects that reference it using a foreign key
- `onDeleteSetNull`: when the object is deleted, set the referencing foreign keys to null.

In the next chapter we will see how you can specify these foreign key constraints also for foreign keys that double as primary key.

7.6. Specifying a natural primary key that is also a foreign key

Let's define a class `UserInfo` which provides additional data for a `User`. We will only allow exactly one `UserInfo` object per `User`, and therefore chose as primary key for the `UserInfo` a reference to the `User`.

Using a foreign key as primary key (tutorial7.C).

```
#include <Wt/Dbo/Dbo>
#include <Wt/Dbo/backend/Sqlite3>

namespace dbo = Wt::Dbo;

class UserInfo;
class User;

namespace Wt {
    namespace Dbo {

        template<>
        struct dbo_traits<UserInfo> : public dbo_default_traits {
            typedef ptr<User> IdType;

            static IdType invalidId() {
                return ptr<User>();
            }

            static const char *surrogateIdField() { return 0; }
        };
    }
}

class User
{
public:
    std::string name;

    dbo::collection< dbo::ptr<UserInfo> > infos;
```

A hands-on introduction to Wt::Dbo

```
template<class Action>
void persist(Action& a)
{
    dbo::field(a, name, "name");

    // In fact, this is really constrained to hasOne() ...
    dbo::hasMany(a, infos, dbo::ManyToOne, "user");
}
};

class UserInfo
{
public:
    dbo::ptr<User> user;
    std::string info;

    template<class Action>
    void persist(Action& a)
    {
        dbo::id(a, user, "user", dbo::OnDeleteCascade);
        dbo::field(a, info, "info");
    }
};

void run()
{
    /*
     * Setup a session, would typically be done once at application startup.
     */
    dbo::backend::Sqlite3 sqlite3(":memory:");
    sqlite3.setProperty("show-queries", "true");
    dbo::Session session;
    session.setConnection(sqlite3);

    session.mapClass<User>("user");
    session.mapClass<UserInfo>("user_info");

    /*
     * Try to create the schema (will fail if already exists).
     */
    session.createTables();

    dbo::Transaction transaction(session);

    {
        User *user = new User();
        user->name = "Joe";

        dbo::ptr<User> userPtr = session.add(user);

        UserInfo *userInfo = new UserInfo();
        userInfo->user = userPtr;
        userInfo->info = "great guy";

        session.add(userInfo);

        transaction.commit();
    }
}
```

```
{
    dbo::Transaction transaction(session);

    dbo::ptr<UserInfo> info = session.find<UserInfo>();

    std::cerr << info->user->name << " is a " << info->info << std::endl;

    transaction.commit();
}

int main(int argc, char **argv)
{
    run();
}
```

As you can see, in this example we would really need a One-to-One relationship, but this currently not yet supported in Dbo and thus we emulate it using a Many-to-One relationship (which has the same representation in SQL).

When run, this should output:

```
begin transaction
create table "user" (
    "id" integer primary key autoincrement,
    "version" integer not null,
    "name" text not null
)

create table "user_info" (
    "version" integer not null,
    "user_id" bigint,
    "info" text not null,
    primary key ("user_id"),
    constraint "fk_user_info_user" foreign key ("user_id")
        references "user" ("id") on delete cascade
)

commit transaction
begin transaction
insert into "user" ("version", "name") values (?, ?)
insert into "user_info" ("version", "user_id", "info") values (?, ?, ?)
commit transaction
begin transaction
select version, "user_id", "info" from "user_info"
select "version", "name" from "user" where "id" = ?
Joe is a great guy
commit transaction
```

8. Transactions and concurrency

Reading data from the database or flushing changes to the database require an active transaction. A Transaction is a RIIA (Resource-Initialization-is-Acquisition) class which at the same time provides isolation between concurrent sessions and atomicity for persisting changes to the database.

The library implements optimistic locking, which allows detection (rather than avoidance) of concurrent modifications. It is a recommended and widely used strategy for dealing with concurrency issues in a scalable manner as no write locks are needed on the database. To detect a concurrent modification, a `version` field is added to each table which is incremented on each modification. When performing a modification (such as updating or removing an object), it is checked that the version of the record in the database is the same as the version of the object that was originally read from the database.

Note

The minimum level of isolation which is required for the library's *optimistic locking* strategy is *Read Committed*: modifications in a transaction are only visible to other sessions as soon as they are committed. This is usually the lowest level of isolation supported by a database.

The `Transaction` class is a light-weight proxy that references a *logical* transaction: multiple (usually nested) `Transaction` objects may be instantiated simultaneously, which each need to be committed for the logical transaction to be committed. In this way you can easily protect individual methods which require database access with such a transaction object, which will automatically participate in a wider transaction if that is available. A transaction will in fact defer opening a real transaction in the database until needed, and thus there is no penalty for instantiating a transaction to make sure a unit of work is atomic, even if you are not yet sure that there will be actual work done.

Transactions may fail and dealing with failing transactions is an integral aspect of their usage. When the library detects a concurrent modification, a `StaleObjectException` is thrown. Other exceptions may be thrown, including exceptions in the backend driver when for example the database schema is not compatible with the mapping. There may also be problems detected by the business logic which may raise an exception and cause the transaction to be rolled back. When a transaction is rolled back, the modified database objects are not successfully synchronized with the database, but may possibly be synchronized later in a new transaction.

Obviously, many exceptions will be fatal. One notable exception is the `StaleObjectException` however. Different strategies are possible to deal with this exception. Regardless of the approach, you will at least need to `reread()` the stale database object(s) before being able to commit changes made in a new transaction.

9. Installation

`Wt::Dbo` is included in `Wt`, and can thus be installed as part of this library for which there may be standard packages available for your operating system.

The library does however in no way depend on `Wt`, and can also be built, installed and used separately from it. Starting from a `Wt` source package (and on in a UNIX-like environment), you would do the following to build and install only `Wt::Dbo`:

Installing `Wt::Dbo` from source (UNIX-like).

```
$ cd wt-xxx
$ mkdir build
$ cd build
```

A hands-on introduction to Wt::Dbo

```
$ cmake ../ # extra options may be needed, see instructions
$ cd src/Wt/Dbo
$ make
$ sudo make install
```

See also the Wt installation instructions [<http://www.webtoolkit.eu/wt/doc/reference/html/InstallationUnix.html>].