Connecting to a Database Engine

Querying the Database

Transactions

Parameterized Queries

CRUD (But With a Lot of Style)

The Smart SQL ORM

Caveat for SQL Tables

Mapper Data Status

Beyond CRUD

Navigation and Pagination

Virtual Fields

Seek and You Shall Find

Profiling

Sometimes It Just Ain't Enough

Pros and Cons

Databases

Connecting to a Database Engine

Fat-Free is designed to make the job of interfacing with SQL databases a breeze. If you're not the type to immerse yourself in details about SQL, but lean more towards object-oriented data handling, you can go directly to the next section of this tutorial. However, if you need to do some complex data-handling and database performance optimization tasks, SQL is the way to go.

Establishing communication with a SQL engine like MySQL, SQLite, PostgreSQL, SQL Server, Sybase, and Oracle is done using the familiar `$f3->set()` command. Connecting to a SQLite database would be:

```
$f3->set('DB', new DB\SQL('sqlite:/absolute/path/to/your/database.sqlite'));
```

You can now work with database object from anywhere in your application with `$f3->get('DB')->exec('…');` .

Another example, this time with MySQL:

```
$db=new DB\SQL(
    'mysql:host=localhost;port=3306;dbname=mysqldb',
    'admin',
    'p455w0rD'
);
```

You can also watch a video (https://youtu.be/dZ_FscVVm3I) that goes over using MySQL in the Fat-Free Framework.

Querying the Database

OK. That was easy, wasn't it? That's pretty much how you would do the same thing in ordinary PHP. You just need to know the DSN format of the database you're connecting to. See the PDO section of the PHP manual (http://www.php.net/manual/en/pdo.connections.php).

Let's continue our PHP code:

```
$f3->set('result',$db->exec('SELECT brandName FROM wherever'));
echo Template::instance()->render('abc.htm');
```

Huh, what's going on here? Shouldn't we be setting up things like PDOs, statements, cursors, etc.? The simple answer is: you don't have to. F3 simplifies everything by taking care of all the hard work in the backend.

This time we create an HTML template like `abc.htm` that has at a minimum the following:

```
<repeat group="{{ @result }}" value="{{ @item }}">
    <span>{{ @item.brandName  }}</span>
</repeat>
```

In most instances, the SQL command set should be enough to generate a Web-ready result so you can use the `result` array variable in your template directly. Be that as it may, Fat-Free will not stop you from getting into its SQL handler internals. In fact, F3's `DB\SQL` class derives directly from PHP's `PDO` class, so you still have access to the underlying PDO components and primitives involved in each process, if you need some fine-grain control.

Transactions

Here's another example. Instead of a single statement provided as an argument to the `$db->exec()` command, you can also pass an array of SQL statements:

```
$db->exec(
    array(
        'DELETE FROM diet WHERE food="cola"',
        'INSERT INTO diet (food) VALUES ("carrot")',
        'SELECT * FROM diet'
    )
);
```

F3 is smart enough to know that if you're passing an array of SQL instructions, this indicates a SQL batch transaction. You don't have to worry about SQL rollbacks and commits because the framework will automatically revert to the initial state of the database if any error occurs during the transaction. If successful, F3 commits all changes made to the database.

You can also start and end a transaction programmatically:

```
$db->begin();
$db->exec('DELETE FROM diet WHERE food="cola"');
$db->exec('INSERT INTO diet (food) VALUES ("carrot")');
$db->exec('SELECT * FROM diet');
$db->commit();
```

A rollback will occur if any of the statements encounter an error.

To get a list of all database instructions issued:

```
echo $db->log();
```

Parameterized Queries

Passing string arguments to SQL statements is fraught with danger. Consider this:

```
$db->exec(
    'SELECT * FROM users '.
    'WHERE username="'.$f3->get('POST.userID').'"'
);
```

If the `POST` variable `userID` does not go through any data sanitation process, a malicious user can pass the following string and damage your database irreversibly:

```
admin"; DELETE FROM users; SELECT "1
```

Luckily, parameterized queries help you mitigate these risks:

```
$db->exec(
    'SELECT * FROM users WHERE userID=?',
    $f3->get('POST.userID')
);
```

If F3 detects that the value of the query parameter/token is a string, the underlying data access layer escapes the string and adds quotes as necessary.

Our example in the previous section will be a lot safer from SQL injection if written this way:

```
$db->exec(
    array(
        'DELETE FROM diet WHERE food=:name',
        'INSERT INTO diet (food) VALUES (?)',
        'SELECT * FROM diet'
    ),
    array(
        array(':name'=>'cola'),
        array(1=>'carrot'),
        NULL
    )
);
```

You can also watch a video (https://youtu.be/55RcfqzEvWM) that goes over SQL Injection and database security.

CRUD (But With a Lot of Style)

F3 is packed with easy-to-use object-relational mappers (ORMs) that sit between your application and your data - making it a lot easier and faster for you to write programs that handle common data operations - like creating, retrieving, updating, and deleting (CRUD) information from SQL and NoSQL databases. Data mappers do most of the work by mapping PHP object interactions to the corresponding backend queries.

Suppose you have an existing MySQL database containing a table of users of your application. (SQLite, PostgreSQL, SQL Server, Sybase will do just as well.) It would have been created using the following SQL command:

```
CREATE TABLE users (
    userID VARCHAR(30),
    password VARCHAR(30),
    visits INT,
    PRIMARY KEY(userID)
);
```

Note: MongoDB is a NoSQL database engine and inherently schema-less. F3 has its own fast and lightweight NoSQL implementation called Jig, which uses PHP-serialized or JSON-encoded flat files. These abstraction layers require no rigid data structures. Fields may vary from one record to another. They can also be defined or dropped on the fly.

Now back to SQL. First, we establish communication with our database.

```
$db=new DB\SQL(
    'mysql:host=localhost;port=3306;dbname=mysqldb',
    'admin',
    'wh4t3v3r'
);
```

To retrieve a record from our table:

```
$user=new DB\SQL\Mapper($db,'users');
$user->load(array('userID=?','tarzan'));
```

The first line instantiates a data mapper object that interacts with the `users` table in our database. Behind the scene, F3 retrieves the structure of the `users` table and determines which field(s) are defined as primary key(s). At this point, the mapper object does not contain any data yet (it is called **"dry state"**) and the `$user` var is basically nothing more than a structured object - but it contains the methods it needs to perform the basic CRUD operations plus some extras as you will see later. Now, to retrieve a record from our `users` table with, e.g., the field `userID` containing the string value `tarzan`, we use the `load()` method. This process is called "auto-hydrating" the data mapper object.

Easy, wasn't it? F3 understands that a SQL table already has a structural definition existing within the database engine itself. Unlike other frameworks, F3 requires no extra class declarations (unless you want to extend the data mappers to fit complex objects), no redundant PHP array/object property-to-field mappings (duplication of efforts), no code generators (which require code regeneration if the database structure changes), no stupid XML/YAML files to configure your models, no superfluous commands just to retrieve a single record. With F3, a simple resizing of a `varchar` field in your MySQL table does not require a single change in your application code. Consistent with MVC and "separation of concerns", the database admin has as much control over the data and the structures as a template designer has over HTML/XML templates.

If you prefer working with NoSQL databases, the similarities in query syntax are superficial. In the case of the MongoDB data mapper, the equivalent code would be:

```
$db=new DB\Mongo('mongodb://localhost:27017','testdb');
$user=new DB\Mongo\Mapper($db,'users');
$user->load(array('userID'=>'tarzan'));
```

With Jig, the syntax is similar to F3's template engine:

```
$db=new DB\Jig('db/data/',DB\Jig::FORMAT_JSON);
$user=new DB\Jig\Mapper($db,'users');
$user->load(array('@userID=?','tarzan'));
```

The Smart SQL ORM

The framework automatically maps the field `visits` in our table to a data mapper property during object instantiation, i.e. `$user=new DB\SQL\Mapper($db,'users');`. Once the object is created, `$user->password` and `$user->userID` would map to the `password` and `userID` fields in our table, respectively.

You can't add or delete a mapped field, or change a table structure using the ORM. You must do this in MySQL, or whatever database engine you're using. After you've made the changes in your database engine, Fat-Free will automatically synchronize the new table structure with your data mapper object when you run your application.

F3 derives the data mapper structure directly from the database schema. No guesswork involved. It understands the differences between MySQL, SQLite, MSSQL, Sybase, and PostgreSQL database engines.

> **Notice:** SQL identifiers should not use reserved words, and should be limited to alphanumeric characters `A-Z`, `0-9`, and the underscore symbol (`_`). Column names containing spaces (or special characters) and surrounded

by quotes in the data definition are not compatible with the ORM. They cannot be represented properly as PHP object properties.

Let's say we want to increment the user's number of visits and update the corresponding record in our `users` table, we can add the following code:

```
$user->visits++;
$user->save();
```

If we wanted to insert a record, we follow this process:

```
$user=new DB\SQL\Mapper($db,'users');
// or $user=new DB\Mongo\Mapper($db,'users');
// or $user=new DB\Jig\Mapper($db,'users');
$user->userID='jane';
$user->password=md5('secret');
$user->visits=0;
$user->save();
```

We still use the same `save()` method. But how does F3 know when a record should be inserted or updated? At the time a data mapper object is auto-hydrated by a record retrieval, the framework keeps track of the record's primary keys (or `_id`, in the case of MongoDB and Jig) - so it knows which record should be updated or deleted - even when the values of the primary keys are changed. A programmatically-hydrated data mapper - the values of which were not retrieved from the database, but populated by the application - will not have any memory of previous values in its primary keys. The same applies to MongoDB and Jig, but using object `_id` as reference. So, when we instantiated the `$user` object above and populated its properties with values from our program - without at all retrieving a record from the user table, F3 knows that it should insert this record.

A mapper object will not be empty after a `save()`. If you wish to add a new record to your database, you must first flush the mapper using the `reset` method:

```
$user->reset();
$user->userID='cheetah';
$user->password=md5('unknown');
$user->save();
```

Calling `save()` a second time without invoking `reset()` will simply update the record currently pointed to by the mapper.

There is also a helpful video available that covers some of the major portions of models/mappers in Fat-Free. Click here (https://youtu.be/qAo7Hpptn3w) to view the video.

Caveat for SQL Tables

Although the issue of having primary keys in all tables in your database is argumentative, F3 does not stop you from creating a data mapper object that communicates with a table containing no primary keys. The only drawback is: you can't delete or update a mapped record because there's absolutely no

way for F3 to determine which record you're referring to plus the fact that positional references are not reliable. Row IDs are not portable across different SQL engines and may not be returned by the PHP database driver.

To remove a mapped record from our table, invoke the `erase()` method on an auto-hydrated data mapper. For example:

```
$user=new DB\SQL\Mapper($db,'users');
$user->load(array('userID=? AND password=?','cheetah','ch1mp'));
$user->erase();
```

Jig's query syntax would be slightly similar:

```
$user=new DB\Jig\Mapper($db,'users');
$user->load(array('@userID=? AND @password=?','cheetah','chimp'));
$user->erase();
```

And the MongoDB equivalent would be:

```
$user=new DB\Mongo\Mapper($db,'users');
$user->load(array('userID'=>'cheetah','password'=>'chimp'));
$user->erase();
```

Mapper Data Status

To find out whether our data mapper was loaded with a valid data record or not, use the `dry` method:

```
if ($user->dry())
    echo 'No record matching criteria';
```

Beyond CRUD

We've covered the CRUD handlers. There are some extra methods that you might find useful:

```
$f3->set('user',new DB\SQL\Mapper($db,'users'));
$f3->get('user')->copyFrom('POST');
$f3->get('user')->save();
```

Notice that we can also use Fat-Free variables as containers for mapper objects. The `copyFrom()` method hydrates the mapper object with elements from a framework array variable, the array keys of which must have names identical to the mapper object properties, which in turn correspond to the record's field names. So, when a Web form is submitted (assuming the HTML name attribute is set to `userID`), the contents of that input field is transferred to `$_POST['userID']`, duplicated by F3 in its `POST.userID` variable, and saved to the mapped field `$user->userID` in the database. The process becomes very simple if they all have identically-named elements. Consistency in array keys, i.e. template token names, framework variable names and field names is key :)

> **Danger:** By default, `copyfrom` takes the whole array provided. This may open a security leak if the user posts more fields than you expect. Use the 2nd parameter to setup a filter callback function to get rid of unwanted

fields to copy from.

On the other hand, if we wanted to retrieve a record and copy the field values to a framework variable for later use, like template rendering:

```
$f3->set('user',new DB\SQL\Mapper($db,'users'));
$f3->get('user')->load(array('userID=?','jane'));
$f3->get('user')->copyTo('POST');
```

We can then assign {{ @POST.userID }} to the same input field's value attribute. To sum up, the HTML input field will look like this:

```
<input type="text" name="userID" value="{{ @POST.userID }}">
```

The `save()`, `update()`, `copyFrom()` data mapper methods and the parameterized variants of `load()` and `erase()` are safe from SQL injection.

Navigation and Pagination

By default, a data mapper's `load()` method retrieves only the first record that matches the specified criteria. If you have more than one that meets the same condition as the first record loaded, you can use the `skip()` method for navigation:

```
$user=new DB\SQL\Mapper($db,'users');
$user->load('visits>3');
// Rewritten as a parameterized query
$user->load(array('visits>?',3));

// For MongoDB users:
// $user=new DB\Mongo\Mapper($db,'users');
// $user->load(array('visits'=>array('$gt'=>3)));

// If you prefer Jig:
// $user=new DB\Jig\Mapper($db,'users');
// $user->load('@visits>?',3);

// Display the userID of the first record that matches the criteria
echo $user->userID;
// Go to the next record that matches the same criteria
$user->skip(); // Same as $user->skip(1);
// Back to the first record
$user->skip(-1);
// Move three records forward
$user->skip(3);
```

You may use `$user->next()` as a substitute for `$user->skip()`, and `$user->prev()` if you think it gives more meaning to `$user->skip(-1)`.

Use the `dry()` method to check if you've maneuvered beyond the limits of the result set. `dry()` will return TRUE if you try `skip(-1)` on the first record. It will also return TRUE if you `skip(1)` on the last record that meets the retrieval criteria.

The `load()` method accepts a second argument: an array of options containing key-value pairs such as:

```
$user->load(
    array('visits>?',3),
    array(
        'order'=>'userID DESC',
        'offset'=>5,
        'limit'=>3
    )
);
```

If you're using MySQL, the query translates to:

```
SELECT * FROM users
WHERE visits>3
ORDER BY userID DESC
LIMIT 3 OFFSET 5;
```

This is one way of presenting data in small chunks. Here's another way of paginating results:

```
$page=$user->paginate(2,5,array('visits>?',3));
```

In the above scenario, F3 will retrieve records that match the criteria `'visits>3'`. It will then limit the results to 5 records (per page) starting at page offset 2 (0-based). The framework will return an array consisting of the following elements:

```
[subset] array of mapper objects that match the criteria
[total] sum of all records for all pages
[limit] same value as the size parameter (here 5)
[count] number of of subsets/pages available
[pos] actual subset position
```

The actual subset position returned will be NULL if the first argument of `paginate()` is a negative number or exceeeds the number of subsets found.

Virtual Fields

There are instances when you need to retrieve a computed value of a field, or a cross-referenced value from another table. Enter virtual fields. The SQL mini-ORM allows you to work on data derived from existing fields.

Suppose we have the following table defined as:

```
CREATE TABLE products (
    productID VARCHAR(30),
    description VARCHAR(255),
    supplierID VARCHAR(30),
    unitprice DECIMAL(10,2),
    quantity INT,
    PRIMARY KEY(productID)
);
```

No `totalprice` field exists, so we can tell the framework to request from the database engine the arithmetic product of the two fields:

```
$item=new DB\SQL\Mapper($db,'products');
$item->totalprice='unitprice*quantity';
$item->load(array('productID=:pid',':pid'=>'apple'));
echo $item->totalprice;
```

The above code snippet defines a virtual field called `totalprice` which is computed by multiplying `unitprice` by the `quantity`. The SQL mapper saves that rule/formula, so when the time comes to retrieve the record from the database, we can use the virtual field like a regular mapped field.

You can have more complex virtual fields:

```
$item->mostNumber='MAX(quantity)';
$item->load();
echo $item->mostNumber;
```

This time the framework retrieves the product with the highest quantity (notice the `load()` method does not define any criteria, so all records in the table will be processed). Of course, the virtual field `mostNumber` will still give you the right figure if you wish to limit the expression to a specific group of records that match a specified criteria.

You can also derive a value from another table:

```
$item->supplierName=
    'SELECT name FROM suppliers '.
    'WHERE products.supplierID=suppliers.supplierID';
$item->load();
echo $item->supplierName;
```

Every time you load a record from the products table, the ORM cross-references the `supplerID` in the `products` table with the `supplierID` in the `suppliers` table.

To destroy a virtual field, use `unset($item->totalPrice);`. The `isset($item->totalPrice)` expression returns TRUE if the `totalPrice` virtual field was defined, or FALSE if otherwise.

Remember that a virtual field must be defined prior to data retrieval. The ORM does not perform the actual computation, nor the derivation of results from another table. It is the database engine that does all the hard work.

Seek and You Shall Find

If you have no need for record-by-record navigation, you can retrieve an entire batch of records in one shot:

```
$frequentUsers=$user->find(array('visits>?',3),array('order'=>'userID'));
```

Jig mapper's query syntax has a slight resemblance:

```
$frequentUsers=$user->find(array('@visits>?',3),array('order'=>'userID'));
```

The equivalent code using the MongoDB mapper:

```
$frequentUsers=$user->find(array('visits'=>array('$gt'=>3)), array('order'=>array('us
erID'=>1)));
```

The `find()` method searches the `users` table for records that match the criteria, sorts the result by `userID` and returns the result as an array of mapper objects. `find('visits>3')` is different from `load('visits>3')`. The latter refers to the current `$user` object. `find()` does not have any effect on `skip()`.

> **Important:**
> Declaring an empty condition, NULL, or a zero-length string as the first argument of `find()` or `load()` will retrieve **all** records. Be sure you know what you're doing - you might exceed PHP's `memory_limit` on large tables or collections

The `find()` method has the following syntax:

```
find(
    $criteria,
    array(
        'group'=>'foo',
        'order'=>'foo,bar',
        'limit'=>5,
        'offset'=>0
    )
);
```

find() returns an array of objects. Each object is a mapper to a record that matches the specified criteria.:

```
$place=new DB\SQL\Mapper($db,'places');
$list=$place->find('state="New York"');
foreach ($list as $obj)
    echo $obj->city.', '.$obj->country;
```

If you need to convert a mapper object to an associative array, use the `cast()` method:

```
$array=$place->cast();
echo $array['city'].', '.$array['country'];
```

To retrieve the number of records in a table that match a certain condition, use the `count()` method.

```
if (!$user->count(array('visits>?',10)))
    echo 'We need a better ad campaign!';
```

There's also a `select()` method that's similar to `find()` but provides more fine-grained control over fields returned. It has a SQL-like syntax:

```
select(
    'foo, bar, baz',
    'foo > ?',
    array(
        'group'=>'foo, bar',
        'order'=>'baz ASC',
        'limit'=>5,
        'offset'=>3
    )
);
```

Much like the `find()` method, `select()` does not alter the mapper object's contents. It only serves as a convenience method for querying a mapped table. The return value of both methods is an array of mapper objects. Using `dry()` to determine whether a record was found by an of these methods is inappropriate. If no records match the `find()` or `select()` criteria, the return value is an empty array.

> **Keep in mind:**
> `load()` hydrates the current mapper object, `findone` returns a new hydrated mapper object, and `find` returns an array of hydrated mapper objects.

> **Danger:** The $options array for is not using parameterized field values and is not sanitized from user inputs. If you simply put GET or POST values into group, order, limit or offset without validating them beforehand, you are opening a security issue and bad things can happen.

Profiling

If you ever want to find out which SQL statements issued directly by your application (or indirectly thru mapper objects) are causing performance bottlenecks, you can do so with a simple:

```
echo $db->log();
```

F3 keeps track of all commands issued to the underlying SQL database driver, as well as the time it takes for each statement to complete - just the right information you need to tweak application performance.

Sometimes It Just Ain't Enough

In most cases, you can live by the comforts given by the data mapper methods we've discussed so far. If you need the framework to do some heavy-duty work, you can extend the SQL mapper by declaring your own classes with custom methods - but you can't avoid getting your hands greasy on some hardcore SQL:

```
class Vendor extends DB\SQL\Mapper {
    // Instantiate mapper
    function __construct(DB\SQL $db) {
        // This is where the mapper and DB structure synchronization occurs
        parent::__construct($db,'vendors');
    }

    // Specialized query
    function listByCity() {
        return $this->select('vendorID,name,city',null,array('order'=>'city DESC'));
        /*
        We could have done the same thing with plain vanilla SQL:
        return $this->db->exec(
            'SELECT vendorID,name,city FROM vendors '.
            'ORDER BY city DESC;'
        );
        */
    }
}

$vendor=new Vendor;
$vendor->listByCity();
```

Extending the data mappers in this fashion is an easy way to construct your application's DB-related models.

Pros and Cons

If you're handy with SQL, you'd probably say: everything in the ORM can be handled with old-school SQL queries. Indeed. We can do without the additional event listeners by using database triggers and stored procedures. We can accomplish relational queries with joined tables. The ORM is just unnecessary overhead. But the point is - data mappers give you the added functionality of using objects to represent database entities. As a developer, you can write code faster and be more productive. The resulting program will be cleaner, if not shorter. But you'll have to weigh the benefits against the compromise in speed - specially when handling large and complex data stores. Remember, all ORMS - no matter how thin they are - will always be just another abstraction layer. They still have to pass the work to the underlying SQL engines.

By design, F3's ORMs do not provide methods for directly connecting objects to each other, i.e. SQL joins - because this opens up a can of worms. It makes your application more complex than it should be, and there's the tendency of objects thru eager or lazy fetching techniques to be deadlocked and even out of sync due to object inheritance and polymorphism (impedance mismatch) with the database entities they're mapped to. There are indirect ways of doing it in the SQL mapper, using virtual fields - but you'll have to do this programmatically and at your own risk.

If you are tempted to apply "pure" OOP concepts in your application to represent all your data (because "everything is an object"), keep in mind that data almost always lives longer than the application. Your program may already be outdated long before the data has lost its value. Don't add another layer of complexity in your program by using intertwined objects and classes that deviate too much from the schema and physical structure of the data.

Before you weave multiple objects together in your application to manipulate the underlying tables in your database, think about this: creating views to represent relationships and triggers to define object behavior in the database engine are more efficient. Relational database engines are designed to handle views, joined tables and triggers. They are not dumb data stores. Tables joined in a view will appear as a single table, and Fat-Free can auto-map a view just as well as a regular table. Replicating JOINs as relational objects in PHP is slower compared to the database engine's machine code, relational algebra and optimization logic. Besides, joining tables repeatedly in our application is a sure sign that the database design needs to be audited, and views considered an integral part of data retrieval. If a table cross-references data from another table frequently, consider normalizing your structures or creating a view instead. Then create a mapper object to auto-map that view. It's faster and requires less effort.

Consider this SQL view created inside your database engine:

```
CREATE VIEW combined AS
    SELECT
        projects.project_id AS project,
        users.name AS name
    FROM projects
    LEFT OUTER JOIN users ON
        projects.project_id=users.project_id AND
        projects.user_id=users.user_id;
```

Your application code becomes simple because it does not have to maintain two mapper objects (one for the projects table and another for users) just to retrieve data from two joined tables:

```
$combined=new DB\SQL\Mapper($db,'combined');
$combined->load(array('project=?',123));
echo $combined->name;
```

Tip:Use the tools as they're designed for. Fat-Free already has an easy-to-use SQL helper. Use it if you need a bigger hammer :) Try to seek a balance between convenience and performance. SQL will always be your fallback if you're working on complex and legacy data structures.

← 4. Views and Templates (views-and-templates)

6. Plug-Ins → (plug-ins)