

Matisse[®] PHP

Programmer's Guide

February 2012



MATISSE PHP Programmer's Guide

Copyright ©1992–2012 Matisse Software Inc. All Rights Reserved.

This manual and the software described in it are copyrighted. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without prior written consent of Matisse Software Inc. This manual and the software described in it are provided under the terms of a license between Matisse Software Inc. and the recipient, and their use is subject to the terms of that license.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. and international patents.

TRADEMARKS: Matisse and the Matisse logo are registered trademarks of Matisse Software Inc. All other trademarks belong to their respective owners.

PDF generated 13 February 2012

1	Introduction	5
	Scope of This Document	5
	Before Reading This Document	5
	Before Running the Examples	5
2	Connection and Transaction	7
	Building the Examples	7
	Read Write Transaction	7
	Read-Only Access	8
	Version Access	8
	Specific Options	10
	More about MtDatabase	12
3	Working with Objects and Values	13
	Running the Examples on Objects	13
	Creating Objects	13
	Listing Objects	15
	Deleting Objects	16
	Comparing Objects	17
	Running the Examples on Values	17
	Setting and Getting Values	17
	Removing Values	18
	Streaming Values	19
	Retrieving an Object from its Oid	20
4	Working with Relationships	21
	Running the Examples on Relationships	21
	Setting and Getting Relationship Elements	21
	Adding and Removing Relationship Elements	22
	Listing Relationship Elements	22
	Counting Relationship Elements	23
5	Working with Indexes	24
	Running the Examples on Indexes	24
	Index Lookup	24
	Index Lookup Count	25
	Index Entries Count	25
6	Working with Entry-Point Dictionaries	26
	Running the Examples on Dictionaries	26
	Entry-Point Dictionary Lookup	26
	Entry-Point Dictionary Lookup Count	27
7	Working with SQL	28
	Running the Examples on SQL	28
	Executing a SQL Statement	28
	Creating Objects	29

Updating Objects	29
Retrieving Values	30
Retrieving Objects from a SELECT statement	31
Retrieving Objects from a Block Statement	32
Executing DDL Statements	33
Executing SQL Methods	34
Deleting Objects	36
8 Working with Class Reflection	38
Running the Examples on Reflection	38
Creating Objects	38
Listing Objects	39
Working with Indexes	40
Working with Entry Point Dictionaries	41
Discovering Object Properties	42
Adding Classes	43
Deleting Objects	44
Removing Classes	44
9 Working with Database Events	46
Running the Events Example	46
Events Subscription	46
Events Notification	47
More about MtEvent	47
10 Handling Namespaces	48
Connection with Factory	48
Creating your Object Factory	48
11 Building your Application	51
Discovering the Matisse PHP Classes	51
Generating Stub Classes	51
Extending the generated Stub Classes	51
Generated Public Methods	53

1 Introduction

Scope of This Document

This document is intended to help PHP programmers learn the aspects of Matisse design and programming that are unique to the Matisse PHP binding.

Aspects of Matisse programming that the PHP binding shares with other interfaces, such as basic concepts and schema design, are covered in *Getting Started with Matisse*.

Future releases of this document will add more advanced topics. If there is anything you would like to see added, or if you have any questions about or corrections to this document, please send e-mail to support@matisse.com.

Before Reading This Document

Throughout this document, we presume that you already know the basics of PHP programming and either relational or object-oriented database design, and that you have read the relevant sections of *Getting Started with Matisse*.

Before Running the Examples

Before running the following examples, you must do the following:

- Install Matisse 9.0.0 or later.
- Install the PHP version 5.3.2 or later for your operating system (a free download from www.php.net).
- Download and extract the Matisse PHP binding source code and sample code from the Matisse Web site:

```
http://www.matisse.com/developers/documentation/
```

The sample code files are grouped in subdirectories by chapter number. For example, the code snippets from the following chapter are in the `chap_2` directory.

- Build the Matisse PHP binding from the source code. Follow the building instructions as detailed in the `BUILD` file.
- Create and initialize a database. You can simply start the Matisse Enterprise Manager, select the database 'example' and right click on 'Re-Initialize'.
- From a Unix shell prompt or on MS Windows from a 'Command Prompt' window, change to the `chap_x` subdirectory in the directory where you installed the examples.
- If applicable, load the ODL file into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema'. For example you may import `chap_3/objects.odl` for the Chapter 3 demo.

- **Generate PHP class files:**

```
mt_sdl stubgen -lang php objects.odl
```

- **Run the application. For instance in `chap_3`:**

```
php -c ../../php.ini createObjects.php host database
```

2 Connection and Transaction

All interaction between client PHP applications and Matisse databases takes place within the context of transactions (either explicit or implicit) established by database connections, which are transient instances of the `MtDatabase` class. Once the connection is established, your PHP application may interact with the database using the schema-specific methods generated by `mt_sdl`. The following sample code shows a variety of ways of connecting with a Matisse database.

Note that in this chapter there is no ODL file as you do not need to create an application schema.

Building the Examples

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_2` directory in your installation (under `examples`).
3. Launch the application:

```
Windows:
php -c ../../php.ini connect.php host database

UNIX:
php -c ../../php.ini connect.php host database
```

Read Write Transaction

The following code connects to a database, starts a transaction, commits the transaction, and closes the connection:

```
try {
    $db = new \matisse\MtDatabase($hostname, $dbname);

    $db->open();

    $db->startTransaction();

    echo "Connection and read-write access to $db\n";

    $db->commit();
    $db->close();
} catch (\matisse\MtException $mtex) {
    echo "Error:". $mtex->getMessage(). "\n";
    echo "Error code:". $mtex->getErrorCode(). "\n";
}
```

1. Launch the application:

```
Windows:
php -c ../../php.ini connect.php host database

UNIX:
php -c ../../php.ini connect.php host database
```

Read-Only Access

The following code connects to a database in read-only mode, suitable for reports:

```
try {
    $db = new \matisse\MtDatabase($hostname, $dbname);
    $db->open();

    echo "Connection and read-only access to $db\n";

    $db->startVersionAccess();

    $db->endVersionAccess();

    $db->close();

    print "Completed\n";
} catch (\matisse\MtException $mtex) {
    echo "Error:". $mtex->getMessage()."\n";
    echo "Error code:". $mtex->getErrorCode()."\n";
}
```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini versionConnect.php host database
```

UNIX:

```
php -c ../../php.ini versionConnect.php host database
```

Version Access

The following code illustrates methods of accessing various versions of a database.

```
function listVersions1($db) {
    $it = $db->versionIterator();
    foreach ($it as $pos => $ver) {
        $vtime = $db->getVersionFromName($ver);
        echo "{$pos} - {$ver} ({$vtime})\n";
    }
}

function listVersions2($db) {
    $it = $db->versionIterator();
    while($it->valid()) {
        print "{$it->key()} - {$it->current()}\n";
        $it->next();
    }
    $it->close();
}

function listVersions3($db) {
    $it = $db->versionIterator();
    while($it->valid()) {
        $it->next();
    }
}
```

```

}
$it->rewind();
while($it->valid()) {
    print "{$it->key()} - {$it->current()}\n";
    $it->next();
}
$it->close();
}

function removeVersions($db) {
    $it = $db->versionIterator();
    foreach ($it as $pos => $ver) {
        $vtime = $db->getVersionFromName($ver);
        echo "{$pos} - {$ver} ({$vtime})\n";
        $db->removeVersion($ver);
    }
}

if ((!isset($argv[1])) || (!isset($argv[2]))) {
    echo "Error: Need to specify arguments: $argv[0] localhost dbname\n";
    exit();
} else {
    require_once("matisse.php");

    $hostname = $argv[1];
    $dbname = $argv[2];

    try {
        $db = new matisse\MtDatabase($hostname, $dbname);
        $db->open();

        echo "Current version: ".$db->getCurrentVersion()."\n";

        $db->startTransaction();
        echo "Version list before regular commit:\n";
        listVersions1($db);
        $db->commit();

        $db->startTransaction();
        echo "Version list after regular commit:\n";
        listVersions2($db);
        $versionName = $db->commit("mysnapshot-");

        $db->startVersionAccess();
        echo "Version list after named commit:\n";
        listVersions3($db);
        $db->endVersionAccess();

        $db->startVersionAccess($versionName);
        echo "Successful access to version: $versionName\n";
        $db->endVersionAccess();

        $db->startTransaction();
        removeVersions($db);
        $db->commit();

        $db->close();
    }
}

```

```

    print "Completed\n";
} catch (matisse\MtException $mtex) {
    echo "Error:". $mtex->getMessage(). "\n";
}
}

```

1. Launch the application:

Windows:

```
php -c ../../php.ini versionNavigation.php host database
```

UNIX:

```
php -c ../../php.ini versionNavigation.php host database
```

Specific Options

This example shows how to enable the local client-server memory transport and to set or read various connection options and states.

```

function startAccess($db, $readonly) {
    if ($readonly) {
        $db->startVersionAccess();
        echo "read-only access to $db\n";
    } else {
        $db->startTransaction();
        echo "read-write access to $db\n";
    }
}

function endAccess($db) {
    if ($db->isVersionAccessInProgress()) {
        $db->endVersionAccess();
        echo "version access ended\n";
    } else if ($db->isTransactionInProgress()) {
        $db->commit();
        echo "transaction committed\n";
    } else {
        echo "No transaction nor version access in progress\n";
    }
}

function isReadOnlyAccess($db) {
    return ($db->getOption(matisse\MtDatabase::DATA_ACCESS_MODE) ==
            matisse\MtDatabase::DATA_READONLY);
}

function setAccessMode($db, $mode) {
    switch ($mode) {
        case "T":
            $db->setOption(matisse\MtDatabase::DATA_ACCESS_MODE,
                matisse\MtDatabase::DATA_MODIFICATION);
            echo "DATA_MODIFICATION (read-write transaction) mode\n";
            break;
        case "V":

```

```

$db->setOption(matisse\MtDatabase::DATA_ACCESS_MODE,
              matisse\MtDatabase::DATA_READONLY);
echo "DATA_READONLY (version) mode\n";
break;
case "S":
    $db->setOption(matisse\MtDatabase::DATA_ACCESS_MODE,
                  matisse\MtDatabase::DATA_DEFINITION);
    echo "DATA_DEFINITION (schema definition) mode\n";
    break;
default:
    echo "unknown mode\n";
    break;
}
}

if ((!isset($argv[1])) || (!isset($argv[2])) || (!isset($argv[3]))) {
    echo "Error: Need to specify arguments: $argv[0] localhost dbname V|T|S\n";
    exit();
} else {
    require_once("matisse.php");

    $hostname = $argv[1];
    $dbname = $argv[2];
    $mode = $argv[3];
    try {
        $db = new matisse\MtDatabase($hostname, $dbname);
        $db->open();

        setAccessMode($db, $mode);

        startAccess($db, isReadOnlyAccess($db));

        echo "\ndo something...\n\n";

        endAccess($db);

        print "Completed\n";
    } catch (matisse\MtException $mtex) {
        echo "Error: ".$mtex->getMessage()."\n";
    }
}
}

```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini advancedConnect.php host database
```

UNIX:

```
php -c ../../php.ini advancedConnect.php host database
```

More about MtDatabase

As illustrated by the previous sections, the `MtDatabase` class provides all the methods for database connections and transactions. The reference documentation for the `MtDatabase` class is included in the Matisse PHP Binding API documentation located from the Matisse PHP binding installation root directory in `docs/php/api/index.html`.

3 Working with Objects and Values

This chapter explains how to manipulate object with the object interface of the Matisse PHP binding. The object interface allows you to directly retrieve objects from the Matisse database without Object-Relational mapping, navigate from one object to another through the relationship defined between them, and update properties of objects without writing SQL statements.

The object interface can be used with Matisse PHP SQL interface as well. For example, you can retrieve objects with SQL, then use the object interface to navigate to other objects from these objects, or update properties of these objects using the accessor methods defined on these classes.

Running the Examples on Objects

This sample program creates objects from 2 classes (`Person` and `Employee`), lists all `Person` objects (which includes both objects, since `Employee` is a subclass of `Person`), deletes objects, then lists all `Person` objects again to show the deletion. Note that because `FirstName` and `LastName` are not nullable, they *must* be set when creating an object.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_3` directory in your installation (under `examples`).
3. Load `objects.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_3/objects.odl` for this demo.
4. Generate PHP class files:

```
mt_sdl stubgen -lang php objects.odl
```

Creating Objects

This section illustrates the creation of objects. The stubclass provides a default constructor which is the base factory for creating persistent objects.

```
public static function createPerson($db) {
    return new Person(self::getClass($db));
}
```

You can also use the default constructor defined on the class.

```
/**
 * Factory constructor. This constructor is called by <code>MtObjectFactory</code>.
 * It is public for technical reasons but is not intended to be called
 * directly by user methods.
 * Cascaded constructor, used by subclasses to create a new object in the database.
 * It is protected for technical reasons but is not intended to be called
 * directly by user methods.
 * @param cls a class descriptor (the class to instantiate)
 * @param db a database
 * @param mtOid an existing object ID in the database
 */
```

```

public function Person($cls, $db=null, $mtOid=0) {
    parent::__construct($cls, $db, $mtOid);
}

// Create a new Person object (instance of class Person)
// use the predefined factory method
$p = Person::createPerson($db);
$p->setFirstName("John");
$p->setLastName("Smith");
$p->setAge(42);

$a = new PostalAddress(PostalAddress::getClass($db));
$a->setCity("Portland");
$a->setPostalCode("97201");
$p->setAddress($a);

// Create a new Employee object
$e = new Employee(Employee::getClass($db));
$e->setFirstName("Jane");
$e->setLastName("Jones");
// Age is nullable we can leave it unset
$e->setHireDate(new DateTime('2009-11-08'));
// numeric datatype
$e->setSalary("85000.00");

```

1. Launch the application:

Windows:

```
php -c ../../php.ini createObjects.php host database
```

UNIX:

```
php -c ../../php.ini createObjects.php host database
```

If your application need to create a large number of objects all at once, we recommend that you use the `preallocate()` method defined on `MtDatabase` which provide a substantial performance optimization.

```

$db->startTransaction();

// Optimize the objects loading
// Preallocate OIDs so objects can be created in the client workspace
// without requesting any further information from the server
$db->preallocate(DEFAULT_ALLOCATOR_CNT);

for ($i = 1; $i <= SAMPLE_OBJECT_CNT; $i++) {
    // Create a new Employee object
    $e = new Employee(Employee::getClass($db));
    $fname = $fNameSample[rand() % MAX_SAMPLES];
    $lname = $lNameSample[rand() % MAX_SAMPLES];
    $e->setFirstName($fname);
    $e->setLastName($lname);
    $hDate = "".(2000+(rand() % MAX_SAMPLES))."-06-01";
    $e->setHireDate(new DateTime($hDate));
    $salary = $salarySample[rand() % MAX_SAMPLES];
    $e->setSalary($salary);

    $a = new PostalAddress(PostalAddress::getClass($db));
    $addrIdx = rand() % MAX_SAMPLES;
    $a->setCity($addressSample[$addrIdx][0]);
}

```

```

$a->setPostalCode($addressSample[$addrIdx][1]);
$e->setAddress($a);

if ($i % OBJECT_PER_TRAN_CNT == 0) {
    $db->commit();
    $db->startTransaction();
}
// check the remaining number of preallocated objects.
if ($db->numPreallocated() < 2) {
    $db->preallocate(DEFAULT_ALLOCATOR_CNT);
}
}

if ($db->isTransactionInProgress())
    $db->commit();

```

1. Launch the application:

```

Windows:
php -c ..\..\php.ini loadObjects.php host database

UNIX:
php -c ../../php.ini loadObjects.php host database

```

Listing Objects

This section illustrates the enumeration of objects from a class. The `instanceIterator()` static method defined on a generated stubclass allows you to enumerate the instances of this class and its subclasses. The `getInstanceNumber()` method returns the number of instances of this class.

```

// List all Person objects
print "\n". Person::getInstanceNumber($db) ." Person(s) in the database.\n";
print "\n". PostalAddress::getInstanceNumber($db) ." Address(s) in the database.\n";
$iiter = Person::instanceIterator($db);
while($iiter->valid()) {
    $x = $iiter->current();

    print "- {"$x->getFirstName()} {"$x->getLastName()} from ".
        ($x->getAddress() != null ? $x->getAddress()->getCity() : "???" )
        ." is a ". $x->getMtClass()->getMtName() ."\n";
    $iiter->next();
}

$iiter->close();

```

1. Launch the application:

```

Windows:
php -c ..\..\php.ini listObjects.php host database

UNIX:
php -c ../../php.ini listObjects.php host database

```

The `ownInstanceIterator()` static method allows you to enumerate the own instances of a class (excluding its subclasses). The `getOwnInstanceNumber()` method returns the number of instances of a class (excluding its subclasses).

```
// List all Person objects
print "\n". Person::getOwnInstanceNumber($db) ." Person(s) (excluding subclasses) in
the database.\n";

$iter = Person::ownInstanceIterator($db);
while($iter->valid()) {
    $x = $iter->current();

    print "- {"$x->getFirstName()} {"$x->getLastName()} from ".
        ($x->getAddress() != null ? $x->getAddress()->getCity() : "???" )
        ." is a ". $x->getMtClass()->getMtName() ."\n";
    $iter->next();
}

$iter->close();
```

1. Launch the application:

```
Windows:
php -c ../../php.ini listOwnInstances.php host database

UNIX:
php -c ../../php.ini listOwnInstances.php host database
```

Deleting Objects

This section illustrates the removal of objects. The `remove()` method delete an object.

```
// Remove created objects
...
// NOTE: does not remove the object sub-parts
$p->remove();
```

To remove an object and its sub-parts, you need to override the `deepRemove()` method in the subclass to meet your application needs. For example the implementation of `deepRemove()` in the `Person` class that contains a reference to a `PostalAddress` object is as follows:

```
/**
 * Overrides MtObject.deepRemove() to remove the Address object if any.
 */
public function deepRemove() {
    $pAddr = $this->getAddress();
    if ($pAddr != null)
        $pAddr->deepRemove();

    parent::deepRemove();
}

...
$p->deepRemove();
```

1. Launch the application:

```
Windows:
php -c ../../php.ini deleteObjects.php host database
```

```
UNIX:
php -c ../../php.ini deleteObjects.php host database
```

The `removeAllInstances()` method defined on `MtClass` delete all the instances of a class.

```
Person::getClass($db)->removeAllInstances ();
```

1. Launch the application:

```
Windows:
php -c ../../php.ini deleteAllObjects.php host database
```

```
UNIX:
php -c ../../php.ini deleteAllObjects.php host database
```

Comparing Objects

This section illustrates how to compare objects. Persistent objects must be compared with the `equal()` method. You can't compare persistent object with the `===` operator.

```
...
if($p1->equals($p2))
    print("Same objects\n");
```

Running the Examples on Values

This example shows how to get and set values for various Matisse data types including Null values, and how to check if a property of an object is a Null value or not.

This example uses the database created for `Objects Example`. It creates objects, then manipulates its values in various ways.

Setting and Getting Values

This section illustrates the set, update and read object property values. The stubclass provides a set and a get method for each property defined in the class.

```
$e = Employee::createEmployee($db);

// Setting strings
$e->setComment("FirstName, LastName, Age, HireDate & Salary Set");
$e->setFirstName("John");
$e->setLastName("Jones");

// Setting numbers
$e->setAge(42);
```

```
// Setting Date (use DateTime class for Date and Timestamp)
$e->setHireDate(new DateTime('2009-11-08'));

// Setting Numeric (int, double or string)
$e->setSalary("74250.00");
$e->setSalary(74250);
$e->setSalary(74250.00);

// Setting an attribute of type int to NULL
$e->setNull(Employee::getAgeAttribute($db));
```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini setObjectValues.php host database
```

UNIX:

```
php -c ../../php.ini setObjectValues.php host database
```

```
// Getting String values
print "\nComment: {$e->getComment()}\n";

print "- {$e->getFirstName()} {$e->getLastName()} is a ". $e->getMtClass()-
>getMtName() ."\n";

// suppresses output if no value set
if (!$e->isAgeNull())
    print " {$e->getAge()} years old\n";
// Getting number values
print " Number of dependents: {$e->getDependents()}\n";
// Getting numeric values
print " Salary: {$e->getSalary()}\n";
// Getting date values
print " Hiring Date: ". $e->getHireDate()->format("Y-m-d") ."\n";
```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini getObjectValues.php host database
```

UNIX:

```
php -c ../../php.ini getObjectValues.php host database
```

Removing Values

This section illustrates the removal of object property values. Removing the value of an attribute will return the attribute to its default value.

```
// Removing value returns attribute to default
$e->removeAge();

// suppresses output if no value set
```

```

if ( ! $e->isAgeNull() )
    print "  {$e->getAge()} years old\n";
else
    print "  Age: null". ($e->isAgeDefaultValue() ? " (default value)" : "") ."\n";

```

1. Launch the application:

Windows:
 php -c ..\..\php.ini removeObjectValues.php host database

UNIX:
 php -c ../../php.ini removeObjectValues.php host database

Streaming Values

This section illustrates the streaming of blob-type values (`MT_BYTES`, `MT_AUDIO`, `MT_IMAGE`, `MT_VIDEO`). The subclass provides streaming methods (`setPhotoElements()`, `getPhotoElements()`) for each blob-type property defined in the class. It also provides a method (`getPhotoSize()`) to retrieve the blob size without reading it.

```

// Setting blobs

// set to 512 for demo purpose
// a few Mega-bytes would be more appropriate
// for real multimedia objects (audio, video, high-resolution photos)
$bufSize = 512;
$handle = @fopen('matisse.gif', "r");
if ($handle) {
    // reset the stream
    $buf = array();
    $e->setPhotoElements($buf, \matisse\reflect\MtType::BEGIN_OFFSET, 0, true);
    while (!feof($handle)) {
        $buf = fread($handle, $bufSize);
        $num = strlen($buf);
        if ($num > 0)
            $e->setPhotoElements($buf, \matisse\reflect\MtType::CURRENT_OFFSET, $num,
false);
    }
    fclose($handle);
}
print "Image of {$e->getPhotoSize()} bytes stored.\n";

print "Streaming an image out...\n";
// Getting blobs (save value of e.Photo as out.gif in the
// program directory)
$bufSize = 512;
$total = 0;
$handle = @fopen('out.gif', "w+");
if ($handle) {
    // reset the stream
    $buf = null;
    $e->getPhotoElements($buf, \matisse\reflect\MtType::BEGIN_OFFSET, 0);
    do {

```

```

        $num = $e->getPhotoElements($buf, \matisse\reflect\MtType::CURRENT_OFFSET,
$bufSize);
        if ($num > 0) {
            $num = fwrite($handle, $buf);
        }
        $total += $num;
    } while ($num == $bufSize);
}
fclose($handle);

print "Image of {$total} bytes read.\n";

```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini readWriteStreamingValues.php host database
```

UNIX:

```
php -c ../../php.ini readWriteStreamingValues.php host database
```

Retrieving an Object from its Oid

This section illustrates a very commonly used feature in the binding. Using the Object Identifier (OID) is very efficient for retrieving one object from the database. The example below illustrates how to view an image stored into the database using the object Identifier to quickly retrieve the object.

```

$db = new \matisse\MtDatabase($hostname, $dbname);

$db->open();

$db->startVersionAccess();
if (isset($_GET['photoid'])) {
    // image request, send out the image
    $photoid = intval($_GET['photoid']);

    $p = $db->upcast($photoid);

    $bytes = $p->getPhoto();
    $db->endVersionAccess();
    $db->close();
    header("Content-type: image/gif");
    echo $bytes;
    flush ();
    exit ();
}

```

4 Working with Relationships

One of the major advantages of the object interface of the Matisse PHP binding is the ability to navigate from one object to another through a relationship defined between them. Relationship navigation is as easy as accessing an object property.

Running the Examples on Relationships

This example creates several objects, then manipulates the relationships among them in various ways.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_4` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_4/examples.odl` for this demo.
4. Generate PHP class files:

```
mt_sdl stubgen -lang php examples.odl
```

Setting and Getting Relationship Elements

This section illustrates the set, update and get object relationship values. The stubclass provides a set and a get method for each relationship defined in the class.

```
$m1 = Manager::createManager($db);
...
// Set a relationship
// Need to report to someone since the relationship
// cardinality minimum is set to 1
$m1->setReportsTo($m1);

$e = Employee::createEmployee($db);
...
// Set a relationship
$e->setReportsTo($m1);

$c1 = Person::createPerson($db);
...
$c2 = Person::createPerson($db);
...
// Set successors
$m1->setChildren( array($c1, $c2) );

...
// Get all successors
$c = $m1->getChildren();
```

1. Launch the application:

```
Windows:
php -c ../../php.ini setRelationships.php host database
```

```
UNIX:
php -c ../../php.ini setRelationships.php host database
```

Adding and Removing Relationship Elements

This section illustrates the adding and removing of relationship elements. The stubclass provides a `append`, a `remove` and a `clear` method for each relationship defined in the class.

```
$c2 = Person::createPerson($db);
$c3 = Person::createPerson($db);
...
// add one successor
$m2->appendChildren($c2);
// add multiple successors
$m2->appendChildren(array($c3));
...
// removing successors (this only breaks links, it does not
// remove objects)
$m2->removeChildren(array($c2));
// removing one successor
$m2->removeChildren($c3);

// clearing all successors (this only breaks links, it does
// not remove objects)
$m2->clearChildren();
```

1. Launch the application:

```
Windows:
php -c ../../php.ini addToRelationship.php host database
php -c ../../php.ini removeFromRelationship.php host database
```

```
UNIX:
php -c ../../php.ini addToRelationship.php host database
php -c ../../php.ini removeFromRelationship.php host database
```

Listing Relationship Elements

This section illustrates the listing of relationship elements for one-to-many relationships. The stubclass provides an iterator method for each one-to-many relationship defined in the class.

```
// Iterate when the relationship is large is always more efficient
$iter = $m2->childrenIterator();

while($iter->valid()) {
    $x = $iter->current();
    print " {$x->getFirstName()}";

    $iter->next();
}
```

```
print "\n";

$iter->close();
```

1. Launch the application:

```
Windows:
php -c ..\..\php.ini iterateRelationship.php host database

UNIX:
php -c ../../php.ini iterateRelationship.php host database
```

Counting Relationship Elements

This section illustrates the counting of relationship elements for one-to-many relationships. The stubclass provides an get size method for each one-to-many relationship defined in the class.

```
// Get the relationship size without loading the PHP objects
// which is the fast way to get the size
$childrenCnt = $m2->getChildrenSize();

print "Relationship size without loading:\n";
print "  {$m2->getFirstName()} has {$childrenCnt} kid(s).\n";

// an alternative to get the relationship size
// but the PHP objects are loaded before you can get the count
$childrenCnt = count($m2->getChildren());

print "Relationship size from the loaded array:\n";
print "  {$m2->getFirstName()} has {$childrenCnt} kid(s).\n";
```

1. Launch the application:

```
Windows:
php -c ..\..\php.ini getRelationshipSize.php host database

UNIX:
php -c ../../php.ini getRelationshipSize.php host database
```

5 Working with Indexes

While indexes are used mostly by the SQL query optimizer to speed up queries, the Matisse PHP binding also provides the index query APIs to look up objects based on a key value(s). The stubclass defines both lookup methods and iterator methods for each index defined on the class.

Running the Examples on Indexes

Using the `PersonName` index, it checks whether the database contains an entry for a person matching the specified name. The application will list the names in the database, indicate whether the specified name was found, and return results within a sample range (defined in the source) using an iterator.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_5` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_5/examples.odl` for this demo.
4. Generate PHP class files:

```
mt_sdl stubgen -lang php examples.odl
```

Index Lookup

This section illustrates retrieving objects from an index. The stubclass provides a lookup and a iterator method for each index defined on the class.

```
// the lookup function returns null to represent no match
$found = Person::lookupPersonName($db, $lastName, $firstName);
```

1. Launch the application:

```
Windows:
php -c ../../php.ini lookupObjects.php host database
```

```
UNIX:
php -c ../../php.ini lookupObjects.php host database
```

```
// open an iterator for a specific range
$fromFirstName = "Fred";
$toFirstName = "John";
$fromLastName = "Jones";
$toLastName = "Murray";
print "\nLookup from \"{$fromFirstName} {$fromLastName}\" to \"{$toFirstName}
{$toLastName}\"";

$iter = Person::personNameIterator($db, $fromLastName, $fromFirstName, $toLastName,
$toFirstName);
```

```

print "\nFound with no class filter:\n";
while($iter->valid()) {
    $p = $iter->current();
    print "  {$p->getFirstName()} {$p->getLastName()}\n";

    $iter->next();
}
$iter->close();

```

1. Launch the application:

Windows:
 php -c ..\..\php.ini iterateIndex.php *host database*

UNIX:
 php -c ../../php.ini iterateIndex.php *host database*

Index Lookup Count

This section illustrates retrieving the object count for a matching index key. The `getObjectNumber()` method is defined on the `MtIndex` class.

```

$key = array( $lastName, $firstName ) ;
$count = Person::getPersonNameIndex($db)->getObjectNumber($key);
print "{$count} objects retrieved\n";

```

1. Launch the application:

Windows:
 php -c ..\..\php.ini lookupObjectsCount.php *host database*

UNIX:
 php -c ../../php.ini lookupObjectsCount.php *host database*

Index Entries Count

This section illustrates retrieving the number of entries in an index. The `getIndexEntriesNumber()` method is defined on the `MtIndex` class.

```

$count = Person::getPersonNameIndex($db)->getIndexEntriesNumber();
print "{$count} entries in the index\n";

```

1. Launch the application:

Windows:
 php -c ..\..\php.ini countIndexEntries.php *host database*

UNIX:
 php -c ../../php.ini countIndexEntries.php *host database*

6 Working with Entry-Point Dictionaries

An entry-point dictionary is an indexing structure containing keywords derived from a value, which is especially useful for full-text indexing. While the entry-point dictionary can be used with SQL query using `ENTRY_POINT` keyword, the object interface of the Matisse PHP binding also provides APIs to directly retrieve objects using the entry-point dictionaries.

Running the Examples on Dictionaries

Using the `commentDict` entry-point dictionary, the example retrieves the `Person` objects in the database with `Comments` fields containing a specified character string.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `chap_6` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `chap_6/examples.odl` for this demo.
4. Generate PHP class files:

```
mt_sdl stubgen -lang php examples.odl
```

Entry-Point Dictionary Lookup

This section illustrates retrieving objects from an entry-point dictionary. The stubclass provides access to lookup methods and iterator methods for each entry-point dictionary defined on the class.

```
// the lookup function returns null to represent no match
// if more than one match an exception is raised
$found = Person::getCommentDictDictionary($db)->lookup($searchstring);
```

1. Launch the application:

```
Windows:
php -c ../../php.ini lookupObjects.php host database
```

```
UNIX:
php -c ../../php.ini lookupObjects.php host database
```

```
$hits = 0;

// open an iterator on the matching person objects
$iter = Person::commentDictIterator($db, $searchstring);
while($iter->valid()) {
    $p = $iter->current();
    print " {$p->getFirstName()} {$p->getLastName()}\n";

    $hits++;
    $iter->next();
}
```

```
print "{$hits} Person(s) with 'comment' containing '{$searchstring}'\n";
```

1. Launch the application:

Windows:

```
php -c ../../php.ini iterateEpDict.php host database
```

UNIX:

```
php -c ../../php.ini iterateEpDict.php host database
```

Entry-Point Dictionary Lookup Count

This section illustrates retrieving the object count for a matching entry-point key. The `getObjectNumber()` method is defined on the `MtEntryPointDictionary` class.

```
$count = Person::getCommentDictDictionary($db)->getObjectNumber($searchstring,
null);
print "{$count} matching object(s) retrieved\n";
```

1. Launch the application:

Windows:

```
php -c ../../php.ini lookupObjectsCount.php host database
```

UNIX:

```
php -c ../../php.ini lookupObjectsCount.php host database
```

7 Working with SQL

Running the Examples on SQL

This sample program demonstrates how to manipulate objects via the Matisse PHP SQL interface. It creates objects (`Person`, `Employee` and `Manager`) and it executes `SELECT` statements to retrieve objects. It also shows how to create SQL methods and execute them.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `SQL` directory in your installation (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `sql/examples.odl` for this demo.
4. Generate PHP class files:

```
mt_sdl stubgen -lang php examples.odl
```

Executing a SQL Statement

After you open a connection to a Matisse database, you can execute statements (i.e., SQL statements or SQL methods) using a `MtStatement` object. You can create a statement object for a specific `MtDatabase` object using the `createStatement` method.

You can create more specific `Statement` objects for different purposes:

- `MtStatement` - It is specifically used for the SQL statements where you don't need to pass any value as a parameter
- `MtPreparedStatement` - It is a subclass of the statement class. The main difference is that, unlike the statement class, prepared statement is compiled and optimized once and can be used multiple times by setting different parameter values.
- `MtCallableStatement` - It provides a way to call a stored procedure on the server from a PHP program. Callable statements also need to be prepared first, and then their parameters are set using the `set` methods.
- `MtResultSet` - It represents a table of data, which is usually generated by executing a statement that queries the database. A `ResultSet` object maintains a cursor pointing to its current row of data.

NOTE: With the Matisse PHP SQL interface you usually don't need to use the PHP stub classes unless you want to retrieve objects from a SQL statement or from the execution of a SQL method.

Creating Objects

You can also create objects into the database without the PHP stub classes. The following code demonstrates how to create multiple objects of the same class using a prepared statement.

```
$db->startTransaction();

// Create an instance of PreparedStatement
$commandText = "INSERT INTO Person (FirstName, LastName, Age) VALUES (?, ?, ?)";
$stmt = $db->prepareStatement($commandText);

// Set parameters
$stmt->setString(1, "James");
$stmt->setString(2, "Watson");
$stmt->setInt(3, 75);

// Execute the INSERT statement
$inserted = $stmt->executeUpdate();

// Set parameters for the next execution
$stmt->setString(1, "Elizabeth");
$stmt->setString(2, "Watson");
$stmt->setNull(3);

// Execute the INSERT statement with new parameters
$inserted = $stmt->executeUpdate();

$db->commit();

// Clean up
$stmt->close();
```

1. Launch the application:

```
Windows:
php -c ..\..\php.ini insertObjects.php host database

UNIX:
php -c ../../php.ini insertObjects.php host database
```

Updating Objects

You can also create objects into the database without the PHP stub classes. The following code demonstrates how to create multiple objects of the same class using a prepared statement.

```
$db->startTransaction();

// Create an instance of Statement
$stmt = $db->createStatement();

// Set the relationship 'Spouse' between these two Person objects
$commandText = "SELECT REF(p) FROM Person p WHERE FirstName = 'James' AND LastName = 'Watson' INTO sp1;";
$stmt->execute($commandText);
```

```

    $commandText = "UPDATE Person SET Spouse = p1 WHERE FirstName = 'Elizabeth' AND
LastName = 'Watson';";
    $inserted = $stmt->executeUpdate($commandText);

    // Clean up
    $stmt->close();

    $db->commit();

```

1. Launch the application:

```

Windows:
php -c ../../php.ini insertObjects.php host database

UNIX:
php -c ../../php.ini insertObjects.php host database

```

Retrieving Values

You use the `ResultSet` object, which is returned by the `executeQuery` method, to retrieve values or objects from the database. Use the `next` method combined with the appropriate `getString`, `getInt`, etc. methods to access each row in the result.

The following code demonstrates how to retrieve string and integer values from a `ResultSet` object after executing a `SELECT` statement.

```

// Create an instance of PreparedStatement
    $commandText = "SELECT FirstName, LastName, Spouse.FirstName AS Spouse, Age FROM
Person WHERE LastName = ? LIMIT 10;";
    $pstmt = $db->prepareStatement($commandText);

// Set parameters
    $pstmt->setString(1, "Watson");

// Execute the SELECT statement and get a ResultSet
    $rset = $pstmt->executeQuery();

print "Total selected: {"$rset->getTotalNumObjects()}\n";
print "Total qualified: {"$rset->getTotalNumQualified()}\n";

// Print column names
    $numberOfColumns = $rset->getColumnCount();

// get the column names; column indexes start from 1
for ($i = 0; $i < $numberOfColumns; $i++) {
    printf("%16s", $rset->getColumnName($i+1));
    print(" ");
}
print("\n");
for ($i = 0; $i < $numberOfColumns; ++$i) {
    print("----- ");
}
print("\n");

// Read rows one by one

```

```

while ($rset->next()) {
    // Get values for the first and second column
    $fname = $rset->getString(1);
    $lname = $rset->getString(2);
    $sfname = $rset->getString(3);
    $age = $rset->getInt(4);
    // The third column 'Age' can be null. Check if it is null or not first.
    if ($rset->wasNull()) $age = "NULL";

    // Print the current row
    printf("%16s %16s %16s ", $fname, $lname, $sfname);
    print("{ $age}\n");
}

// Clean up and close the database connection
$rset->close();
$stmt->close();

```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini selectValues.php host database
```

UNIX:

```
php -c ../../php.ini selectValues.php host database
```

Retrieving Objects from a SELECT statement

You can retrieve PHP objects directly from the database without using the Object-Relational mapping technique. This method eliminates the unnecessary complexity in your application, i.e., O/R mapping layer, and improves your application performance and maintenance.

To retrieve objects, use `REF` in the select-list of the query statement and the `getObject` method returns an object. The following code example shows how to retrieve `Person` objects from a `ResultSet` object.

```

// Set the SELECT statement. Note that we use REF() in the select-list
// to get the PHP objects directly (rather than values)
$commandText = "SELECT REF(p) FROM Person p WHERE LastName = 'Watson'";

$stmt = $db->createStatement();

// Execute the SELECT statement and get a ResultSet
$rset = $stmt->executeQuery($commandText);

print "Total selected:  {"$rset->getTotalNumObjects()}\n";
print "Total qualified: {"$rset->getTotalNumQualified()}\n";
print "Total columns:  {"$rset->getColumnCount()}\n";
print "\n";

printf("  Object Class:      FirstName:      LastName: Spouse FirstName:
Age:\n");

// Read rows one by one

```

```

while ($rset->next()) {
    // Get the Person object
    $p = $rset->getObject(1);

    // Print the current object values
    printf("%16.16s %16s %16s %17s ",
        $p->getMtClass()->getMtName(),
        $p->getFirstName(),
        $p->getLastName(),
        $p->getSpouse()->getFirstName());
    if ($p->getAge() != NULL)
    print("{ $p->getAge() }");
    else
    print("NULL");
    print("\n");
}

// Clean up
$rset->close();
$stmt->close();

```

1. Launch the application:

Windows:

```
php -c ..\..\php.ini selectObjects.php host database
```

UNIX:

```
php -c ../../php.ini selectObjects.php host database
```

Retrieving Objects from a Block Statement

You can also retrieve a collection of PHP objects directly from the database by executing a SQL block statement.

The `getObject` method defined on a `MtCallableStatement` is used to return one object as well as an object collection. The following code example shows how to retrieve a collection of `Person` objects from a `MtCallableStatement`.

```

// Set a block statement
$commandText =
    "BEGIN\n".
    "  DECLARE res SELECTION(Employee);\n".
    "  DECLARE emp_sel SELECTION(Employee);\n".
    "  DECLARE mgr_sel SELECTION(Manager);\n".
    "  SELECT REF(p) FROM ONLY Employee p WHERE p.ReportsTo IS NULL INTO emp_sel;\n".
    "  SELECT REF(p) FROM Manager p WHERE COUNT(p.Team) > 1 INTO mgr_sel;\n".
    "  SET res = SELECTION(emp_sel UNION mgr_sel);\n".
    "  RETURN res;\n".
    "END";

$stmt = $db->prepareCall($commandText);

// Execute a block statement, and get the returned object selection

```

```

$isRset = $stmt->execute();

// Get Result Type
$resultType = $stmt->getResultType();
if (($resultType == \matisse\sql\MtStatement::METHOD) ||
    ($resultType == \matisse\sql\MtStatement::PROCEDURE)) {
    // CALL statement with a return value
    $returnType = $stmt->getParamType(0);

    if ( \matisse\reflect\MtType::SELECTION == $returnType ||
        \matisse\reflect\MtType::OID == $returnType ) {

$sel = $stmt->getObject(0);

print("result Cnt: ". count($sel) ."\n");

foreach ($sel as $e) {
    print("". $e->getMtClass()->getMtName() .": ".
        $e->getFirstName() ." ".
        $e->getLastName() ." - Hiring Date: ".
        $e->getHireDate()->format("Y-m-d") ."\n");
    }
}

}

// Clean up
$stmt->close();

```

1. Launch the application:

```

Windows:
php -c ../../\php.ini insertObjects.php host database

UNIX:
php -c ../../php.ini insertObjects.php host database

```

Executing DDL Statements

You can also create schema objects from a PHP application via SQL.

Creating a Class

You can create schema objects using the `executeUpdate` Method as long as the transaction is started in the `DATA DEFINITION` mode.

```

$db = new \matisse\MtDatabase($hostname, $dbname);
// In order to execute DDL statements, the transaction needs to be
// started in the "Data Definition" mode
$db->setOption (\matisse\MtDatabase::DATA_ACCESS_MODE,
\matisse\MtDatabase::DATA_DEFINITION);
$db->startTransaction();
// Execute the DDL statement
$stmt = $db->createStatement ();
$stmt->executeUpdate ("CREATE CLASS Manager UNDER Employee (bonus INTEGER)");

```

```
$db->commit();
$db->close();
```

Creating a SQL Method

Creating a schema object using the `execute` Method does not require to start a transaction. A transaction will be automatically started in the `DATA DEFINITION` mode.

```
$db->open();

$stmt = $db->createStatement();

// The first method returns the number of Person objects which have a specified
last name
$commandText =
    "CREATE STATIC METHOD CountByLName(lname STRING)\n".
    "RETURNS INTEGER\n".
    "FOR Person\n".
    "BEGIN\n".
    "  DECLARE cnt INTEGER;\n".
    "  SELECT COUNT(*) INTO cnt FROM Person WHERE LastName = lname;\n".
    "  RETURN cnt;\n".
    "END;";

$stmt->execute($commandText);

// Clean up
$stmt->close();

$db->commit();

$db->close();
```

1. Launch the application:

```
Windows:
php -c ../../php.ini createSqlMethod.php host database

UNIX:
php -c ../../php.ini createSqlMethod.php host database
```

Executing SQL Methods

You can call a SQL method using the `CALL` syntax, i.e., simply passing the SQL method name followed by its arguments as an SQL statement. You can also use the `Callable Statement` object, which allows you to explicitly specify the method's parameters.

Executing a Method returning a Value

The following program code shows how to call the SQL method `CountByLName` of the `Person` class.

```
// Specify the stored method. we call a static method,
// the name is consisted of class name and method name.
// Use CALL syntax to call the method
```

```

$commandText = "CALL Person::CountByLName(?)";

// Create an instance of CallableStatement
$stmt = $db->prepareCall($commandText);

// Set parameters
$stmt->setString(1, "Watson");

//Execute the stored method
$stmt->execute();

// Get the returned value
$count = $stmt->getInt(0);

// Print it
print ("{$count} objects found\n");

// Clean up
$stmt->close();

```

1. Launch the application:

```

Windows:
php -c ..\..\php.ini callSqlMethod1.php host database

UNIX:
php -c ../../php.ini callSqlMethod1.php host database

```

Executing a Method returning an Object

The following program code shows how to call the SQL method `FindByName` of the `Person` class.

```

// Specify the SQL method. Since we call a static method,
// the name is consisted of class name and method name.
// Use CALL syntax to call the method
$commandText = "CALL Person::FindByName('Watson', 'James')";

// Create an instance of CallableStatement
$stmt = $db->prepareCall($commandText);

//Execute the stored method
$stmt->execute();

// Get the returned value
$p = $stmt->getObject(0);

// Print it
if ($p)
    print("Found: {$p->getLastName()} {$p->getFirstName()}\n");
else
    print("no matching object found\n");

// Clean up
$stmt->close();

```

1. Launch the application:

```
Windows:
php -c ../../php.ini callSqlMethod2.php host database

UNIX:
php -c ../../php.ini callSqlMethod2.php host database
```

Catching a Method Execution Error

The following program code shows how to retrieve the execution stack trace of a SQL method when an error occurs.

```
try {

    $db = new \matisse\MtDatabase($hostname, $dbname);

    $db->open();
    $db->startVersionAccess();

    $commandText = "CALL Employee::GetAnEmployee()";

    // Create an instance of CallableStatement
    $stmt = $db->prepareCall($commandText);

    //Execute the stored method
    $stmt->execute();

    // Clean up
    $stmt->close();

    $db->endVersionAccess();
    $db->close();

} catch (\matisse\MtException $mtex) {
    $stackTrace = $stmt->getStmtInfo(\matisse\sql\MtStatement::STMT_ERRSTACK);
    print("\nExecution Error in:\n");
    print("{ $stmt->getStmtText() }\n");
    print("\nError Message:\n");
    print("{ $mtex->getMessage() }\n");
    print("\nExecution Stack Trace:\n");
    print("{ $stackTrace }\n");
    print("\nFunction Call Trace:\n");
    print("{ $mtex->getTraceAsString() }\n");
}
```

1. Launch the application:

```
Windows:
php -c ../../php.ini callSqlMethod3.php host database

UNIX:
php -c ../../php.ini callSqlMethod3.php host database
```

Deleting Objects

You can delete objects from the database with a DELETE statement as follows:

```
$db = new \matisse\MtDatabase($hostname, $dbname);

$db->open();

$db->startTransaction();

$stmt = $db->createStatement();

// Delete all the instances of the Person Class
// Execute the DELETE statement
$result = $stmt->executeUpdate("DELETE FROM Person");

// Clean up
$stmt->close();

$db->commit();

$db->close();
```

1. Launch the application:

Windows:

```
php -c ../../php.ini clearPersonObjects.php host database
```

UNIX:

```
php -c ../../php.ini clearPersonObjects.php host database
```

8 Working with Class Reflection

This section illustrates Matisse Reflection mechanism. This example shows how to manipulate persistent objects without having to create the corresponding PHP stubclass. It also presents how to discover all the object properties.

Running the Examples on Reflection

This example creates several objects, then manipulates them to illustrate Matisse Reflection mechanism.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `reflection` directory (under `examples`).
3. Load `examples.odl` into the database. From the Enterprise Manager, select the database 'example' and right click on 'Schema->Import ODL Schema', then select `reflection/examples.odl` for this demo.

Creating Objects

This example shows how to create persistent objects without the corresponding PHP stubclass. The static method `get()` defined on all Matisse Meta-Schema classes (i.e. `MtClass`, `MtAttribute`, etc.) allows you to access to the schema descriptor necessary to create objects. Each object is an instance of the `MtObject` base class. The `MtObject` class holds all the methods to update the object properties (attribute and relationships (i.e. `setString()`, `setSuccessors()`, etc.).

```
// the MtCoreObjectFactory class provides a minimal object factory
// well suited for applications using reflection to manipulate objects
$db = new \matisse\MtDatabase($hostname, $dbname, new MtCoreObjectFactory());
$db->open();
$db->startTransaction();

print "Creating one Person...\n";
// Create a Person object
$pClass = \matisse\reflect\MtClass::get($db, "Person");
$fnAtt = \matisse\reflect\MtAttribute::get($db, "FirstName", $pClass);
$lnAtt = \matisse\reflect\MtAttribute::get($db, "LastName", $pClass);
$cgAtt = \matisse\reflect\MtAttribute::get($db, "collegeGrad", $pClass);
$p = new \matisse\reflect\MtObject($pClass);
$p->setString($fnAtt, "John");
$p->setString($lnAtt, "Smith");
$p->setBoolean($cgAtt, false);

print "Creating one Employee...\n";
// Create a Employee object
$eClass = \matisse\reflect\MtClass::get($db, "Employee");
$hdtAtt = \matisse\reflect\MtAttribute::get($db, "hireDate", $eClass);
$slAtt = \matisse\reflect\MtAttribute::get($db, "salary", $eClass);
$e = new \matisse\reflect\MtObject($eClass);
$e->setString($fnAtt, "James");
$e->setString($lnAtt, "Roberts");
```

```

$e->setDate($hdAtt, new DateTime('2010-01-06'));
$e->setNumeric($slAtt, "5123.25");
$e->setBoolean($cgAtt, true);

print "Creating one Manager...\n";
// Create a Manager object
$mClass = \matisse\reflect\MtClass::get($db, "Manager");
$mRshp = \matisse\reflect\MtRelationship::get($db, "team", $mClass);
$m = new \matisse\reflect\MtObject($mClass);
$m->setString($fnAtt, "Andy");
$m->setString($lnAtt, "Brown");
$m->setDate($hdAtt, new DateTime('2009-11-08'));
$m->setNumeric($slAtt, "7421.25");
$m->setSuccessors($mRshp, array($m, $e));
$m->setBoolean($cgAtt, true);

$db->commit();
$db->close();

```

1. Launch the application:

Windows:

```
php -c ../../php.ini createObjects.php host database
```

UNIX:

```
php -c ../../php.ini createObjects.php host database
```

Listing Objects

This example shows how to list persistent objects without the corresponding PHP stubclass. The `instanceIterator()` method defined on the `MtClass` object allows you to access all instances defined on the class.

```

// List all objects
$pClass = \matisse\reflect\MtClass::get($db, "Person");
$fnAtt = \matisse\reflect\MtAttribute::get($db, "FirstName", $pClass);
$lnAtt = \matisse\reflect\MtAttribute::get($db, "LastName", $pClass);
$cgAtt = \matisse\reflect\MtAttribute::get($db, "collegeGrad", $pClass);

print "\n". $pClass->getInstancesNumber() . " Person(s) in the database.\n";

$iter = $pClass->instanceIterator();

while($iter->valid()) {
    $p = $iter->current();

    print "- ". $p->getMtClass()->getMtName() ." #". $p->getMtOid();
    print " ". $p->getString($fnAtt) ." ". $p->getString($lnAtt);
    print " collegeGrad=". var_export($p->getBoolean($cgAtt, true) ."\n";

    $iter->next();
}

$iter->close();

```

1. Launch the application:

```
Windows:
php -c ../../php.ini listObjects.php host database

UNIX:
php -c ../../php.ini listObjects.php host database
```

Working with Indexes

This example shows how to retrieve persistent objects from an index. The `MtIndex` class holds all the methods retrieves objects from an index key.

```
// List all objects
$pClass = \matisse\reflect\MtClass::get($db, "Person");
$fnAtt = \matisse\reflect\MtAttribute::get($db, "FirstName", $pClass);
$lfnAtt = \matisse\reflect\MtAttribute::get($db, "LastName", $pClass);

// Get the Index Descriptor object
$iClass = \matisse\reflect\MtIndex::get($db, "personName");

// Get the number of entries in the index
$count = $iClass->getIndexEntriesNumber();
print "{$count} entries in the index.\n";

print "Looking for: $firstName $lastName\n";

// lookup for the number of objects matching the key
$key = array( $lastName, $firstName );
$count = $iClass->getObjectNumber($key, null);
print "{$count} matching objects to be retrieved.\n";

if ($count > 1) {
    // More than one matching object
    // Retrieve them with an iterator
    $iter = $iClass->iterator($key, $key);

    while($iter->valid()) {
        $p = $iter->current();
        print " found {$p->getString($fnAtt)} {$p->getString($lfnAtt)} OID={$p->getMtOid()}\n";
        $iter->next();
    }
} else {
    // At most 1 object
    // Retrieve the matching object with the lookup method
    $p = $iClass->lookup($key);
    if ($p != null) {
        print " found {$p->getString($fnAtt)} {$p->getString($lfnAtt)}\n";
    } else {
        print " Nobody found";
    }
}
}
```

1. Launch the application:

```
Windows:
php -c ../../php.ini indexLookup.php host database

UNIX:
php -c ../../php.ini indexLookup.php host database
```

Working with Entry Point Dictionaries

This example shows how to retrieve persistent objects from an Entry Point Dictionary. The `MtEntryPointDictionary` class holds the methods to retrieve objects from a string key.

```
// List all objects
$pClass = matisse\MtClass::get($db, "Person");
$fnAtt = matisse\MtAttribute::get($db, "FirstName", $pClass);
$lnAtt = matisse\MtAttribute::get($db, "LastName", $pClass);
$cgAtt = matisse\MtAttribute::get($db, "collegeGrad", $pClass);

// Get the Index Descriptor object
$sepClass = matisse\MtEntryPointDictionary::get($db, "collegeGradDict");

print "Looking for Persons with CollegeGrad={$collegeGrad}\n";

// lookup for the number of objects matching the key
$count = $sepClass->getObjectNumber($collegeGrad, null);
print "{$count} matching objects to be retrieved.\n";

if ($count > 1) {
    // More than one matching object
    // Retrieve them with an iterator
    $iter = $sepClass->iterator($collegeGrad);
    while($iter->valid()) {
        $p = $iter->current();
        print " found OID={$p->getMtOid()} {$p->getString($fnAtt)} {$p->getString($lnAtt)}";
        print " collegeGrad=". var_export($p->getBoolean($cgAtt), true) ."\n";
        $iter->next();
    }
} else {
    // At most 1 object
    // Retrieve the matching object with the lookup method
    $p = $sepClass->lookup($collegeGrad);
    if ($p != null) {
        print " found OID={$p->getMtOid()} {$p->getString($fnAtt)} {$p->getString($lnAtt)}";
        print " collegeGrad=". var_export($p->getBoolean($cgAtt), true) ."\n";
    } else {
        print " Nobody found\n";
    }
}
```

1. Launch the application:

```
Windows:
php -c ../../php.ini entryPointLookup.php host database
```

```
UNIX:
php -c ../../php.ini entryPointLookup.php host database
```

Discovering Object Properties

This example shows how to list the properties directly from an object. The `MtObject` class holds the `attributesIterator()` method, `relationshipsIterator()` method and `inverseRelationshipsIterator()` method which enumerate the object properties.

```
$iter = $pClass->instancesIterator();

while($iter->valid()) {
    $p = $iter->current();

    print "\n- {$p->getMtClass()->getMtName()} # {$p->getMtOid()}\n";

    print " Attributes:\n";
    $propIter = $p->attributesIterator();

    while($propIter->valid()) {
        $a = $propIter->current();

        $propName = $a->getMtName();
        $propType = $a->getMtType();
        $valType = $p->getType($a);
        $fmtVal = null;
        switch ($valType) {
            case MT_DATE:
                $fmtVal = $p->getDate($a)->format("Y-m-d");
                break;
            case MT_NUMERIC:
                $fmtVal = $p->getNumeric($a);
                break;
            case MT_NULL:
                $fmtVal = 'null';
                break;
            default:
                $fmtVal = $p->getValue($a);
        }
        print "    {$propName} (" . \matisse\reflect\MtType::toString($propType) .
            "):\t " . var_export($fmtVal, true) . " (" .
            \matisse\reflect\MtType::toString($valType) . ") \n";

        $propIter->next();
    }
    $propIter->close();

    print " Relationships:\n";
    $rshpIter = $p->relationshipsIterator();

    while($rshpIter->valid()) {
        $r = $rshpIter->current();

        print "    {$r->getMtName()} :\t {$p->getSuccessorSize($r)} element(s)\n";
    }
}
```

```

$rhsIter->next();
}
$rhsIter->close();

print " Inverse Relationships:\n";
$rhsIter = $p->inverseRelationshipsIterator();
while($rhsIter->valid()) {
$r = $rhsIter->current();

print "    {$r->getMtName()} :\t {$p->getSuccessorSize($r)} element(s)\n";
$rhsIter->next();
}
$rhsIter->close();

$iter->next();
}

$iter->close();

```

1. Launch the application:

Windows:

```
php -c ../../php.ini listObjectProperties.php host database
```

UNIX:

```
php -c ../../php.ini listObjectProperties.php host database
```

Adding Classes

This example shows how to add a new class to the database schema. The connection needs to be open in the DDL (`MtDatabase::DATA_DEFINITION`) mode. Then you need to create instances of `MtClass`, `MtAttribute` and `MtRelationship` and connect them together.

```

$db = new \matisse\MtDatabase($hostname, $dbname, new MtCoreObjectFactory());

// open connection in DDL mode
$db->setOption(\matisse\MtDatabase::DATA_ACCESS_MODE,
\matisse\MtDatabase::DATA_DEFINITION);
$db->open();

$db->startTransaction();

print "Creating 'PostalAddress' class and linking it to 'Person'...\n";

$cAtt = \matisse\reflect\MtAttribute::createAttribute($db, "City",
\matisse\reflect\MtType::STRING);
$pcAtt = \matisse\reflect\MtAttribute::createAttribute($db, "PostalCode",
\matisse\reflect\MtType::STRING);

$paClass = \matisse\reflect\MtClass::createClass($db, "PostalAddress", array( $cAtt,
$pcAtt ), null);

$pClass = \matisse\reflect\MtClass::get($db, "Person");

```

```

    $adRshp = \matisse\reflect\MtRelationship::createRelationship($db, "Address",
    $paClass, array ( 0, 1 ) );
    $pClass->addMtRelationship($adRshp);

    $db->commit();
    $db->close();

```

1. Launch the application:

```

Windows:
php -c ../../php.ini addClass.php host database

UNIX:
php -c ../../php.ini addClass.php host database

```

Deleting Objects

This example shows how to delete persistent objects. The `MtObject` class holds `remove()` and `deepRemove()`. Note that on `MtObject` `deepRemove()` does not execute any cascading delete but only calls `remove()`.

```

$db->startTransaction();

$pClass = \matisse\reflect\MtClass::get($db, "Person");

// Retrieve the object from the previous transaction
$iIter = $pClass->instancesIterator();

while($iIter->valid()) {
    $p = $iIter->current();

    $p->deepRemove();

    $iIter->next();
}

$iIter->close();

$db->commit();

```

1. Launch the application:

```

Windows:
php -c ../../php.ini deleteObjects.php host database

UNIX:
php -c ../../php.ini deleteObjects.php host database

```

Removing Classes

This example shows how to remove a class for the database schema. The `deepRemove()` method defined on `MtClass` will delete the class and its properties and indexes. The connection needs to be open in `MtDatabase.DATA_DEFINITION` mode.

```

$db = new \matisse\MtDatabase($hostname, $dbname, new MtCoreObjectFactory());

    // open connection in DDL mode
    $db->setOption(\matisse\MtDatabase::DATA_ACCESS_MODE,
\matisse\MtDatabase::DATA_DEFINITION);
    $db->open();

    $db->startTransaction();

    $paClass = \matisse\reflect\MtClass::get($db, "PostalAddress");

    print "Removing ". $paClass->getMtClass()->getMtName() .
        " {"$paClass->getMtName()} (#{'$paClass->getMtOid()})\n";

    $paClass->deepRemove();

    $db->commit();
    $db->close();

```

1. Launch the application:

Windows:

```
php -c ../../php.ini removeClass.php host database
```

UNIX:

```
php -c ../../php.ini removeClass.php host database
```

9 Working with Database Events

This section illustrates Matisse Event Notification mechanism. The sample application is divided in two sections. The first section is event selection and notification. The second section is event registration and event handling.

Running the Events Example

This example creates several events, then manipulates them to illustrate the Event Notification mechanism.

1. Follow the instructions in *Before Running the Examples* on page 5.
2. Change to the `events` directory (under `examples`).
3. Launch the application:
To run the example, you need to open 2 command line windows and run one command in each windows.

Windows:

```
php -c ../../php.ini subscribeEvents.php host database
php -c ../../php.ini notifyEvents.php host database
```

UNIX:

```
php -c ../../php.ini subscribeEvents.php host database
php -c ../../php.ini notifyEvents.php host database
```

Events Subscription

This section illustrates event registration and event handling. Matisse provides the `MtEvent` class to manage database events. You can subscribe up to 32 events (`MtEvent.EVENT1` to `MtEvent.EVENT32`) and then wait for the events to be triggered.

```
$TEMPERATURE_CHANGES_EVT = \matisse\MtEvent::EVENT1;
$RAINFALL_CHANGES_EVT = \matisse\MtEvent::EVENT2;
$HIMIDITY_CHANGES_EVT = \matisse\MtEvent::EVENT3;
$WINDSPEED_CHANGES_EVT = \matisse\MtEvent::EVENT4;

$db = new \matisse\MtDatabase($hostname, $dbname);
$db->open();

// Create a subscriber Event
$subscriber = new \matisse\MtEvent($db);

// Subscribe to all 4 events
$eventSet = $TEMPERATURE_CHANGES_EVT |
            $RAINFALL_CHANGES_EVT |
            $HIMIDITY_CHANGES_EVT |
            $WINDSPEED_CHANGES_EVT;
```

```

// Subscribe
$subscriber->subscribe($eventSet);

// Wait 1000 ms for events to be triggered
// return 0 if not event is triggered until the timeout is reached
if (($triggeredEvents = $subscriber->wait(1000)) != 0) {
    print "Events (#{i}) triggered with value {$triggeredEvents}:\n";
    if (($triggeredEvents & $TEMPERATURE_CHANGES_EVT) == 0)
        print "No ";
    print "Change in temperature\n";

    if (($triggeredEvents & $RAINFALL_CHANGES_EVT) == 0)
        print "No ";
    print "Change in rain fall\n";

} else {
    print "No Event received after 1 sec\n";
}

print "Unsubscribe to 4 Events\n";
// Unsubscribe to all 4 events
$subscriber->unsubscribe();

```

Events Notification

This section illustrates event selection and notification.

```

$TEMPERATURE_CHANGES_EVT = \matisse\MtEvent::EVENT1;
$RAINFALL_CHANGES_EVT = \matisse\MtEvent::EVENT2;
$HIMIDITY_CHANGES_EVT = \matisse\MtEvent::EVENT3;
$WINDSPEED_CHANGES_EVT = \matisse\MtEvent::EVENT4;

$db = new \matisse\MtDatabase($hostname, $dbname);
$db->open();

// Create a notifier Event
notifier = new \matisse\MtEvent($db);

$eventSet = $HIMIDITY_CHANGES_EVT;
$eventSet |= $WINDSPEED_CHANGES_EVT;

// Notify of 2 events
$notifier->notify($eventSet);

$db->close();

```

More about MtEvent

As illustrated by the previous sections, the `MtEvent` class provides all the methods for managing database events. The reference documentation for the `MtEvent` class is included in the Matisse PHP Binding API documentation located from the Matisse installation root directory in `docs/php/api/index.html`.

10 Handling Namespaces

You can generate your class stubs in the `company\project\module` PHP namespaces with the `mt_sdl` utility using `-n` option as follows:

```
mt_sdl stubgen -lang php -n 'company\project\module' examples.odl
```

When your persistent classes are defined in a specific namespace, you need to give this information to the `Connection` object so that it can find these classes when returning objects.

Connection with Factory

Using `MtDynamicObjectFactory`

For example, the persistent classes are defined in the `company\project\module` PHP namespace. In this case, you need to pass an `MtDynamicObjectFactory` object as the third argument for the `MtDatabase` constructor.

```
$db = new \matisse\MtDatabase("host", "db", new
    \matisse\MtDynamicObjectFactory(array ("\\company\\project\\module" =>
    "company.project.module")));
```

By default, the anonymous default namespace is searched as well as the specified namespaces.

Using `MtCoreObjectFactory`

This factory is the basic `MtObject`-based object factory. This factory is the most appropriate for application which does use generated stubs. This factory is faster than the default Object Factory used by `MtDatabase` since it doesn't use reflection to build objects.

```
$db = new \matisse\MtDatabase("host", "db", new \matisse\MtCoreObjectFactory());
```

Creating your Object Factory

Implementing the `MtObjectFactory` interface

The `MtObjectFactory` interface describes the mechanism used by `MtDatabase` to create the appropriate PHP object for each Matisse object. Implementing the `MtObjectFactory` interface requires to define the `getPHPClass()` method which returns the PHP class corresponding to a Matisse Class Name, the `getDatabaseClass()` method which returns a Matisse class name corresponding to a PH class name and the `getObjectInstance()` method which returns a PHP object based on an oid.

```
class MyAppFactory implements MtObjectFactory
{
    /**
     * Implements <code>MtObjectFactory.getPHPClass</code>.
     *
     * @param mtClsName      a Matisse Class Name
     * @return MtObject classname
```

```

*/
public function getPHPClass($mtClsName) {
    return "\matisse\reflect\MtObject";
}

/**
 * Implements <code>MtObjectFactory.getDatabaseClass</code>.
 *
 * @param string phpClsName    a PHP Full Class Name
 * @return string              the Matisse Full class name
 */
public function getDatabaseClass($phpClsName) {
    $res = $phpClsName;
    return $res;
}

/**
 * Implements <code>MtObjectFactory.getObjectInstance</code>.
 *
 * @param db a database
 * @param mtOid the OID of the PHP object to create
 * @return the PHP object represented by <code>mtOid</code>
 */
public function getObjectInstance($db, $mtOid) {
    if ($mtOid == 0) {
        return NULL;
    }
    return new \matisse\reflect\MtObject(NULL, $db, $mtOid);
}
}

```

Implementing a Sub-Class of MtCoreObjectFactory

This MtCoreObjectFactory is a basic MtObject-based object factory which can be extended to implement your own Object Factory.

```

class MyOwnObjectFactory implements MtCoreObjectFactory {
{
    /**
     * Implements <code>MtObjectFactory.getPHPClass</code>.
     *
     * @param mtClsName    a Matisse Class Name
     * @return MtObject classname
     */
    public function getPHPClass($mtClsName) {
        // your PHP class name as you see fit
        return $myphpclsname;
    }

    /**
     * Implements <code>MtObjectFactory.getDatabaseClass</code>.
     *
     * @param string phpClsName    a PHP Full Class Name
     * @return string              the Matisse Full class name
     */
    public function getDatabaseClass($phpClsName) {
        $res = $phpClsName;
        // your Matisse class name as you see fit
    }
}

```

```
    return $res;
}

/**
 * Implements <code>MtObjectFactory.getObjectInstance</code>.
 *
 * @param db a database
 * @param mtOid the OID of the PHP object to create
 * @return the PHP object represented by <code>mtOid</code>
 */
public function getObjectInstance($db, $mtOid) {
    if ($this->isSchemaObject($db, $mtOid)) {
        return parent::getObjectInstance($db, $mtOid);
    } else {
        // Create your PHP object as you see fit
        return anObject;
    }
}
}
```

11 Building your Application

This section describes the process for building an application from scratch with the Matisse PHP binding.

Discovering the Matisse PHP Classes

The Matisse PHP library is comprised of in 2 PHP files:

1. **matisse.php** contains all the core classes defined in the **matisse** namespace. These classes manages the database connection, the object factories as well as the objects caching mechanisms. It also includes the Matisse meta-schema classes defined in the **matisse\reflect** namespace.
2. **matisseSql.php** contains all the SQL-related classes defined in the **matisse\sql** namespace. These classes manages the execution of al types of SQL statements.

The Matisse PHP API documentation included in the delivery provides a detailed description of all the classes and methods.

Generating Stub Classes

The PHP binding relies on object-to-object mapping to access objects from the database. Matisse `mt_sdl` utility allows you to generate the stub classes mapping your database schema classes. Generating PHP stub classes is a 2 steps process:

1. Design a database schema using ODL (Object Definition Language).
2. Generate the PHP code from the ODL file:

```
mt_sdl stubgen -lang php myschema.odl
```

A `.php` file will be created for each class defined in the database. If you need to define these persistent classes in a specific namespace, use `-n` option. The following command generates classes under the namespace `MyCompany\MyProject`:

```
mt_sdl stubgen -lang php -n 'MyCompany\MyProject' myschema.odl
```

When you update your database schema later, load the updated schema into the database. Then, execute the `mt_sdl` utility in the directory where you first generated the class files, to update the files. Your own program codes added to these stub class files will be preserved.

Extending the generated Stub Classes

You can add your own source code outside of the `BEGIN` and `END` markers produced in the generated stub class.

```
// BEGIN Matisse SDL Generated Code
// DO NOT MODIFY UNTIL THE 'END of Matisse SDL Generated Code' MARK BELOW
```

```
...  
// END of Matisse SDL Generated Code
```

Appendix A: Generated Public Methods

The following methods are generated automatically in the `.php` class files generated by `mt_sdl`.

For schema classes

The following methods are created for each schema class. These are class methods (also called static methods): that is, they apply to the class as a whole, not to individual instances of the class. These examples are taken from `Person`.

Count instances `getInstanceNumber($db)`
 `getOwnInstanceNumber($db)`

Open an iterator `instanceIterator($db)`
 `ownInstanceIterator($db)`

Sample constructor `createPerson($db)`

Sample toString `__toString()`

Get descriptor `getClass($db)`

Returns an `MtClass` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

Factory constructor `Person($cls, $db=null, $mtKey=0)`

This constructor is called by `MtObjectFactory`. It is public for technical reasons but is not intended to be called directly by user methods.

For all attributes

The following methods are created for each attribute. For example, if the ODL definition for class `Check` contains the attributes `Date` and `Amount`, the `Check.PHP` file will contain the methods `getDate` and `getAmount`. These examples are taken from `Person.firstName`.

Get value `getFirstName()`

Set value `setFirstName($val)`

Remove value `removeFirstName()`

Check Null value `isFirstNameNull()`

Check Default value `isFirstNameDefault()`

Get descriptor `getFirstNameAttribute($db)`

Returns an `MtAttribute` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

For list-type attributes only

The following methods are created for each list-type attribute. These examples are from `Person.photo`.

```

Get elements   getPhotoElements($value, $offset, $len)

Set elements   setPhotoElements($value, $offset, $len, $discardAfter)

Count elements getPhotoSize()

```

For all relationships

The following methods are created for each relationship. These examples are from `Person.spouse`.

```

Clear successors clearSpouse()

Get descriptor   getSpouseRelationship($db)

                    Returns an MtRelationship object. This method supports advanced Matisse
                    programming techniques such as dynamically modifying the schema.

```

For relationships where the maximum cardinality is 1

The following methods are created for each relationship with a maximum cardinality of 1. These examples are from `Manager.assistant`.

```

Get successor   getAssistant()

Set successor   setAssistant($succ)

```

For relationships where the maximum cardinality is greater than 1

The following methods are created for each relationship with a maximum cardinality greater than 1. These examples are from `Manager.team`.

```

Get successors   getTeam()

Open an iterator teamIterator()

Count successors getTeamSize()

Set successors   setTeam($succs)

Add successors  Insert one successor before any existing successors:
                    prependTeam($succ)

                    Add one successor after any existing successors:
                    appendTeam($succ)

                    Add multiple successors after any existing successors:
                    appendTeam($succs)

Remove           removeTeam($succ)
successors

```

```
removeTeam($succs)
```

Remove specified successors.

For indexes

The following methods are created for every index defined for a database. These examples are for the only index defined in the example, `Person.personName`.

Lookup `lookupPersonName($db, $lastName, $firstName)`

Open an iterator `personNameIterator($db, $fromLastName, $fromFirstName, $toLastName, $toFirstName)`
`personNameIterator($db, $fromLastName, $fromFirstName, $toLastName, $toFirstName, $filterClass, $direction, $numObjPerBuffer)`

Get descriptor `getPersonNameIndex($db)`

Returns an `MtIndex` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

For entry-point dictionaries

The following methods are created for every entry-point dictionary defined for a database. These examples are for the only dictionary defined in the example, `Person.commentDict`.

Lookup `lookupCommentDict($db, $value)`

Open an iterator `commentDictIterator($db, $value)`
`commentDictIterator($db, $value, $filterClass, $numObjPerBuffer)`

Get descriptor `getCommentDictDictionary($db)`

Returns an `MtEntryPointDictionary` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.