```
connect($dbhost, $dbuser, $dbpass); $d->select_db($dbname); $this->driver = $d; } public
function get_driver() { return $this->driver; } public function save_instance($phi) { // We don't do
an is_dirty check on the instance to save, if a programmer // wants to update even if the instance
wasn't dirty, is up to them. // They are able to check for the dirty too. // TODO: transactional since
it does cascade save of has one. // TODO: will try to do the loop detection on phi_to_data. //$loop
= time()."_". rand()."_". rand(); //$table = $this->phi_to_data($phi); //print_r($table); //echo
'save_instance '. $phi->getClass() . PHP_EOL; if ($phi->get_id() == null) { //echo 'save_instance
save' . PHP_EOL; $id = $this->save_instance_recursive($phi); } else { //echo 'save_instance
update' . PHP_EOL; $id = $this->update_instance_recursive($phi); } return $id; } private function
save_instance_recursive($phi) { //echo 'save_instance_recursive '. $phi->getClass() . PHP_EOL;
$id = null; // can be an empty array derived from a null has one // this wont save anything so wont
return an id //if (count($table) == 0) return null; // 1. save/update has ones first to save the FK in
the current object $hones = $phi->getAllHasOne(); foreach ($hones as $attr=>$value) { // if the
object is null, then the FK should be set to null // TODO: it would be better to do it on the Phi->set()
if ($value == null) { $phi->set($attr.'_id', null); // FK should be emptied } else { // the has one might
be already saved, if not => save, else => update if ($value->get_id() == null) // insert, is always
dirty if not saved { $idho = $this->save_instance_recursive($value); $value->set_id($idho); // id
set on associated instance $phi->set($attr .'_id', $idho); // FK set on owner } else { $idho = $value-
>get_id(); if ($value->get_is_dirty()) // update has_one only if it's dirty { $this-
>update_instance_recursive($value); } //$idho = $this->update_instance_recursive($value); $phi-
>set($attr .'_id', $idho); // FK set on owner } } } // 2. save the current object $table = $this-
>phi_to_data($phi); //var_dump($table); //print_r($table); /* // save foreigns first to get their fk ids
and set them to the table before save if (count($table['foreigns']) > 0) { // $col is equals to the
name of the field declared in the phi foreach ($table['foreigns'] as $col => $ft) { // if the has one
already exists, do not insert if (!isset($table['columns'][$col .'_id'])) { $id = $this-
>save_instance_recursive($ft); $ft->set_id($id); $table['columns'][$col .'_id'] = $id; // FK set } } } */
// insert with column values $insert_query = $this->table_to_insert($table);
//print_r($insert_query); $r = $this->driver->execute($insert_query); if($r === 1) { $id = $this-
>driver->last_insert_id(); // 3. save the has manies one_to_many, to store the backlink to the
current phi->id // FIXME: missing check for one_to_many and many_to_many $hmanies = $phi-
>getAllHasMany(); foreach ($hmanies as $attr=>$collection) { foreach ($collection as $item) {
$backlink = $this->backlink_name($phi, $attr); //echo 'backlink name: '. $backlink . PHP_EOL; //
even if the item is not dirty, we need to save/update it because the backlink // could be new and
we don't know if that was updated or not, so here we don't // have an is_dirty check. $item-
>setBacklinkId($phi->getClass(), $attr, $item->getClass(), $backlink, $id); // save or update has
many item if ($item->get_id() == null) { $hmid = $this->save_instance_recursive($item); $item-
>set_id($hmid); } else { $this->update_instance_recursive($item); } } } /* // now that I have the id,
save the hasmany in one to many relationships, // injecting the id in the backlink. if
(count($table['many_back']) > 0) { foreach ($table['many_back'] as $backlink_name =>
$manyhmtable) { foreach ($manyhmtable as $i=>$hmtable) { $hmtable['columns']
[$backlink_name] = $id; $hmid = $this->save_instance_recursive($hmtable); //$ft->set_id($hmid);
} } } */ // 4. // TODO: save many to many in join table, might need to detect the owner side $phi-
```

```
>set_id($id); } return $id; } private function update_instance_recursive($phi) { $id = $phi-
>get_id(); // 1. save/update has ones first to save the FK in the current object $hones = $phi-
>getAllHasOne(); foreach ($hones as $attr=>$value) { // if the object is null, then the FK should be
set to null // TODO: it would be better to do it on the Phi->set() if ($value == null) { $phi-
>set($attr.'_id', null); // FK should be emptied } else { // the has one might be already saved, if not
=> save, else => update if ($value->get_id() == null) // insert { $idho = $this-
>save_instance_recursive($value); $value->set_id($idho); // id set on associated instance $phi-
>set($attr .'_id', $idho); // FK set on owner } else { $idho = $value->get_id(); if ($value-
>get_is_dirty()) // update has_one only if it's dirty { $this->update_instance_recursive($value); }
//$idho = $this->update_instance_recursive($value); $phi->set($attr .'_id', $idho); // FK set on
owner } } } $table = $this->phi_to_data($phi); $update_query = $this->table_to_update($table); $r
= $this->driver->execute($update_query); // $r will be 0 if all the values are the same as the one s
in the database // when that happens, we still need to execute the code below //if($r === 1) //{
$hmanies = $phi->getAllHasMany(); foreach ($hmanies as $attr=>$collection) { foreach
($collection as $item) { $backlink = $this->backlink_name($phi, $attr); //echo 'backlink name: '.
$backlink . PHP_EOL; $item->setBacklinkId($phi->getClass(), $attr, $item->getClass(),
$backlink, $id); // save or update has many item if ($item->get_id() == null) { $hmid = $this-
>save_instance_recursive($item); $item->set_id($hmid); } else { $this-
>update_instance_recursive($item); } } } //} return $id; } public function delete_instance($phi) { if
($phi->id == null) throw new \Exception("Instance can't be deleted, it is not yet saved to the
database"); $table_name = $this->get_table_name($phi); $r = $this->driver->query('DELETE
FROM '. $table_name .' WHERE id='. $phi->id); if ($r == 0) { throw new \Exception("Couldn't
delete the instance"); } } /** * Retrieves the phersistent instance from the database. * @param
string $class_name class name with namespace * @param int @id id of the instance in the
database */ public function get_instance($class_name, $id) { $parts = explode('\\', $class_name);
$class = $parts[count($parts)-1]; //$phi = $GLOBALS[$class]->create(); $table_name = $this-
>get_table_name_ph($GLOBALS[$class]); try { $table = $this->get_row($table_name, $id);
//print_r($table); // the row class chould be the same as $class_name or a subclass, si we need //
to use the specific class in the row to create the right instance here $class = $this-
>full_class_name_to_simple_name($table['columns']['class']); $phi = $GLOBALS[$class]-
>create(); $phi->setProperties($table['columns']); //print_r($phi); $phi->set_id($table['columns']
['id']); $phi->set_class($table['columns']['class']); $phi->set_deleted($table['columns']['deleted']);
$phi->set_is_dirty(false); } catch (\Exception $e) { //echo $e->getMessage() . PHP_EOL; return
null; // row doesnt exists, null phi is returned } return $phi; } public function count($class_name) {
$parts = explode('\\', $class_name); $class = $parts[count($parts)-1]; $phi = $GLOBALS[$class]-
>create(); $table_name = $this->get_table_name($phi); $r = $this->driver->query('SELECT
COUNT(id) as count FROM '. $table_name); $row = $r->fetch_assoc(); $r->close(); return
$row['count']; } public function list_instances($class_name, $max, $offset, $sort = 'id', $order =
'asc') { $class = $this->full_class_name_to_simple_name($class_name); $phi =
$GLOBALS[$class]->create(); $table_name = $this->get_table_name($phi); $records = array();
$r = $this->driver->query('SELECT * FROM '. $table_name .' LIMIT '. $offset .', '. $max .' ORDER
BY '. $sort .' '. $order); while ($row = $r->fetch_assoc()) { // FIXME: table is really row or record
```

```
$table = array('table_name' => $table_name, 'columns' => array(), 'foreigns' => array());
$table['columns'] = $row; $records[] = $table; } $r->close(); $instances = array(); foreach($records
as $table) { // the row class chould be the same as $class_name or a subclass, si we need // to
use the specific class in the row to create the right instance here //$phi = $GLOBALS[$class]-
>create(); $class = $this->full_class_name_to_simple_name($table['columns']['class']); $phi =
$GLOBALS[$class]->create(); $phi->setProperties($table['columns']); $phi-
>set_id($table['columns']['id']); $phi->set_class($table['columns']['class']); $phi-
>set_deleted($table['columns']['deleted']); $phi->set_is_dirty(false); $instances[] = $phi; } return
$instances; } /** * Get one_to_many instances. A has_many B as "bs" * $owner_id is A.id,
$class_name is B, $backlink_name is a_bs_back * This loads all the B in A.bs with
backlink_name = owner_id * Who calls this function will assign the results in the correspondent
collection A.bs */ public function list_hasmany_instances($owner_id, $class_name,
$backlink_name) { // Starts equals to list_instances, que query is the difference // Maybe both
functions can be refactored $class = $this->full_class_name_to_simple_name($class_name);
$phi = $GLOBALS[$class]->create(); $table_name = $this->get_table_name($phi); $records =
array(); $r = $this->driver->query('SELECT * FROM '. $table_name .' WHERE '. $backlink_name
.'='. $owner_id); while ($row = $r->fetch_assoc()) { // FIXME: table is really row or record $table =
array('table_name' => $table_name, 'columns' => array(), 'foreigns' => array()); $table['columns']
= $row; $records[] = $table; } $r->close(); $instances = array(); foreach($records as $table) { // the
row class chould be the same as $class_name or a subclass, si we need // to use the specific
class in the row to create the right instance here //$phi = $GLOBALS[$class]->create(); $class =
$this->full_class_name_to_simple_name($table['columns']['class']); $phi = $GLOBALS[$class]-
>create(); $phi->setProperties($table['columns']); $phi->set_id($table['columns']['id']); $phi-
>set_class($table['columns']['class']); $phi->set_deleted($table['columns']['deleted']); $phi-
>set_is_dirty(false); $instances[] = $phi; } return $instances; } // Recursion example: // 0: [ AND =>
[ OR => [c1, c2], c3 ] ] // 1: [ OR => [c1, c2], c3 ] // c3 is processed iterativelly with the OR // 2: [c1,
c2] // // $subtree could have more than one key at the top level, // in that case, should return an
array of expressions generated // from each simple or complex condition, then the recursion
parent // will arrange the expressions in ANDs, ORs, NOTs private function
find_by_where_recursive($subtree, $table_alias) { //var_dump($subtree); $expressions = array();
foreach ($subtree as $k=>$subconds) { if ($k === 'AND' || $k === 'OR') { //echo "COMPLEX
COND".PHP_EOL; // $subcond should be an array of conditions, which can have the same struct
as subtree // binary AND/OR for now $subexprs = $this->find_by_where_recursive($subconds,
$table_alias); $expr = '('; foreach ($subexprs as $subexpr) { $expr .= $subexpr .' '. $k .' '; } //
remove last AND/OR $expr = substr($expr, 0, -(strlen($k)+2)) .')'; $expressions[] = $expr; } else if
($k === 'NOT') { // $subcond could be a simple cond: array(attr, op, value) // or an array with one
root cond being AND, OR, NOT, so it has the same structure as subtree // in both cases subcond
is an array, so need to check for number of items if (count($subconds) == 1) // complex cond,
needs recursion { $subexprs = $this->find_by_where_recursive($subconds, $table_alias); // result
should be one expression $expr = 'NOT '. $subexprs[0]; // already has parenthesis from the
AND/OR expr $expressions[] = $expr; } else // simple cond, count will be 3 { // TODO: same code
as below, please refactor! $refvalue = (isset($subconds[2]) ? $subconds[2] : null); // the refvalue is
null when the operator is "IS NULL" if (is_bool($refvalue)) { $refvalue = ($refvalue ? 'true' : 'false');
} else if (!is_string($refvalue) && is_numeric($refvalue)) // numbers wont come as strings, but
is_numeric returns true for numeric strings also { // NOP } else if (!$refvalue &&
strcasecmp($subconds[1], 'IS NULL') == 0) // a IS NULL { // NOP } else if (!$refvalue &&
strcasecmp($subconds[1], 'IS NOT NULL') == 0) // a IS NOT NULL { // NOP } else if
(strcasecmp($subconds[1], 'MATCH') == 0) // MATCH for FULLTEXT search: MATCH(col)
AGAINST('value') { $expressions[] = 'NOT MATCH('. $table_alias .".". $subconds[0] .') '.
$subconds[2]; continue; } else { $refvalue = "'". addslashes($refvalue) ."'"; } // simple cond render:
alias.col op refvalue $expressions[] = 'NOT '. $table_alias .".". $subconds[0] ." ". $subconds[1] ." ".
$refvalue; } } else // simple condition { //echo "SIMPLE COND".PHP_EOL; $refvalue =
(isset($subconds[2]) ? $subconds[2] : null); // the refvalue is null when the operator is "IS NULL" if
(is_bool($refvalue)) { $refvalue = ($refvalue ? 'true' : 'false'); } else if (!is_string($refvalue) &&
is_numeric($refvalue)) // numbers wont come as strings, but is_numeric returns true for numeric
strings also { // NOP } else if (!$refvalue && strcasecmp($subconds[1], 'IS NULL') == 0) // a IS
NULL { // NOP } else if (!$refvalue && strcasecmp($subconds[1], 'IS NOT NULL') == 0) // a IS
NOT NULL { // NOP } else if (strcasecmp($subconds[1], 'MATCH') == 0) // MATCH for
FULLTEXT search: MATCH(col) AGAINST('value') { $expressions[] = 'MATCH('. $table_alias .".".
$subconds[0] .') '. $subconds[2] .' '; continue; } else { $refvalue = "'". addslashes($refvalue) ."'"; } //
simple cond render: alias.col op refvalue $expressions[] = $table_alias .".". $subconds[0] ." ".
$subconds[1] ." ". $refvalue; } } return $expressions; } /** * Query on one table, this works for
inheritance because for now we have only * single table inheritance. For MTI we need to split the
conditions over multiple * tables and join using the id. */ public function find_by($class_name,
$where, $max, $offset) { $class = $this->full_class_name_to_simple_name($class_name); $phi =
$GLOBALS[$class]->create(); $table_name = $this->get_table_name($phi); $alias =
$table_name[0]; /* $query_where = ""; foreach ($where as $cond) { $refvalue = $cond[2]; if
(is_bool($refvalue)) { $refvalue = ($refvalue ? 'true' : 'false'); } else if (!is_string($refvalue) &&
is_numeric($refvalue)) // numbers wont come as strings, but is_numeric returns true for numeric
strings also { // NOP } else { $refvalue = "'". addslashes($refvalue) ."'"; } $query_where .= $alias .".".
$cond[0] ." ". $cond[1] ." ". $refvalue; $query_where .= " AND "; // FIXME: AND / OR / NOT should
come in the where structure! } $query_where = substr($query_where, 0, -5); // REMOVES THE
LAST AND */ $expr = $this->find_by_where_recursive($where, $alias); $query_where = $expr[0];
$records = array(); $r = $this->driver->query('SELECT * FROM '. $table_name .' as '. $alias .'
WHERE '. $query_where .' LIMIT '. $offset .', '. $max); while ($row = $r->fetch_assoc()) { //
FIXME: table is really row or record $table = array('table_name' => $table_name, 'columns' =>
array(), 'foreigns' => array()); $table['columns'] = $row; $records[] = $table; } $r->close();
$instances = array(); foreach($records as $table) { // the row class chould be the same as
$class_name or a subclass, si we need // to use the specific class in the row to create the right
instance here //$phi = $GLOBALS[$class]->create(); $class = $this-
>full_class_name_to_simple_name($table['columns']['class']); $phi = $GLOBALS[$class]-
>create(); $phi->setProperties($table['columns']); $phi->set_id($table['columns']['id']); $phi-
>set_class($table['columns']['class']); $phi->set_deleted($table['columns']['deleted']); $phi-
>set_is_dirty(false); $instances[] = $phi; } return $instances; } public function
```

```
count_by($class_name, $where) { $class = $this-
>full_class_name_to_simple_name($class_name); $phi = $GLOBALS[$class]->create();
$table_name = $this->get_table_name($phi); $alias = $table_name[0]; $expr = $this-
>find_by_where_recursive($where, $alias); // just generates the criteria from the recursive
struture in $where $query_where = $expr[0]; $records = array(); $r = $this->driver-
>query('SELECT COUNT(id) as count FROM '. $table_name .' as '. $alias .' WHERE '.
$query_where); while ($row = $r->fetch_assoc()) { // FIXME: table is really row or record $table =
array('table_name' => $table_name, 'columns' => array(), 'foreigns' => array()); $table['columns']
= $row; $records[] = $table; } $r->close(); return $records[0]['columns']['count']; } /** * Returns a
table structure if the id exists on the table. */ public function get_row($table_name, $id) { // FIXME:
table is really a recor or row // Does lazy loading, so foreigns are not loaded. $table =
array('table_name' => $table_name, 'columns' => array(), 'foreigns' => array()); $r = $this->driver-
>query('SELECT * FROM '. $table_name .' WHERE id='. $id); if(mysqli_num_rows($r) == 1) {
$row = $r->fetch_assoc(); $table['columns'] = $row; $r->close(); } else { throw new
\Exception('Record with id '. $id .' on table '. $table_name .' does not exist'); } //print_r($table);
return $table; } /** * Takes a table derived from phi and returns an insert query with the column *
values to insert on one single table. */ private function table_to_insert($table) { $columns_string =
''; $values_string = ''; foreach ($table['columns'] as $col => $val) { // id will be set from the auto
increment on the database if ($col == 'id') continue; // if class has namepsace, needs to be
escaped for mysql if ($col == 'class') $val = str_replace('\\', '\\\\', $val); // Check if FK column is
null, do not include on the column list if (\basic\BasicString::endsWith($col, '_id') && $val == null) {
continue; } $columns_string .= $col .', '; // FIXME: need to check the type of value and add or not
quotes depending // on the type, e.g. numeric wont have quotes // TODO: refactor to function
value to db if (is_null($val)) { $values_string .= 'NULL'; } else if (is_bool($val)) { $values_string .=
($val ? 'true' : 'false'); } else if (!is_string($val) && is_numeric($val)) // numbers wont come as
strings, but is_numeric returns true for numeric strings also { $values_string .= $val; } else { if ($val
=== '') $values_string .= 'NULL'; // empty string is NULL in the DB else $values_string .= "'".
addslashes($val) ."'"; } $values_string .= ', '; } $columns_string = substr($columns_string, 0, -2);
$values_string = substr($values_string, 0, -2); $q = 'INSERT INTO '. $table['table_name'] .'('.
$columns_string .') VALUES ('. $values_string .')'; return $q; } private function
table_to_update($table) { $set = ''; foreach ($table['columns'] as $col => $val) { // class and id
wont be updated if ($col == 'class') continue; if ($col == 'id') continue; // TODO: refactor to function
value to db if (is_null($val)) { $val = 'NULL'; } else if (is_bool($val)) { $val = ($val ? 'true' : 'false'); }
else if (!is_string($val) && is_numeric($val)) // numbers wont come as strings, but is_numeric
returns true for numeric strings also { $val = $val; } else { if ($val === '') $val = 'NULL'; // empty
string is NULL in the DB else $val = "'". addslashes($val) ."'"; } $set .= $col .'='. $val .', '; } $set =
substr($set, 0, -2); $q = 'UPDATE '. $table['table_name'] .' SET '. $set .' WHERE id='.
$table['columns']['id']; return $q; } /** * ORM * For now inheritance ORM is all STI. * FIXME: should
detect recursion loops, need to add a model with a loop to test. */ public function
phi_to_data($phi) { // TODO: the returned item should be an array of tables // will contain the amy
table, the associated via has one // and the join tables referencing the main and the assoc table //
saving order will be always main + associated, saving associated first // to copy keys to owners,
```

then join tables, always checking for loops. $table = array(); if ($phi == null) return $table; $table['table_name'] = $this->get_table_name($phi); $table['columns'] = array(); // simple column values $table['foreigns'] = array(); // associated objects referenced by FKs $table['many_back'] = array(); $table['many_join'] = array(); $fields = $phi->getDefinition()->get_all_fields(); foreach ($fields as $field => $type) { if ($field == 'table') continue; // table is a reserved attr name to specify custom table names if ($phi->getDefinition()->is_has_many($field)) // has many { // one to many uses back links from the many side if ($phi->getDefinition()->is_one_to_many($field)) { $backlink_name = $this->backlink_name($phi, $field); $table['many_back'][$backlink_name] = array(); foreach ($phi->get($field) as $i=>$hmphi) { $table['many_back'][$backlink_name][] = $this->phi_to_data($hmphi); // inject backlink name on columns // will be empty until we save the current phi and get an id // when saving the one side, it sets the backlink id to the many side, // so the hmphi should have the id of the container object $table['many_back'][$backlink_name][$i]['columns'][$backlink_name] = $phi->get_id(); } } else // many to many uses join table { //echo 'is NOT one to many '. $field . PHP_EOL; // TBD: manage join table } } else if ($phi->getDefinition()->is_has_one($field)) // has one { // FK field $has_one_field = $field . '_id'; // if has one is not saved, the id will be null // internally PhInstance will set the xxx_id field $table['columns'][$has_one_field] = $phi->get($has_one_field); // creates related table with the has_one value // the associated element can be null $table['foreigns'][$field] = $this->phi_to_data($phi->get($field)); } else if ($phi->getDefinition()->is_serialized_array($field) || $phi->getDefinition()->is_serialized_object($field)) { // addslashes escapes the internal strings in the SQL query // removed the addslashes because it was escaping twice, table_to_insert() already escapes string values $value = $phi->get($field); if ($value != null) $table['columns'][$field] = json_encode($value); //addslashes(json_encode($phi->get($field))); } else // simple field { // test for fields that should not be saved if ($field == 'is_dirty') continue; //echo $field .' '. $type .' is simple field '. PHP_EOL; $table['columns'][$field] = $phi->get($field); } } // columns injected on instances /* $table['columns']['id'] = $phi->get_id(); $table['columns']['deleted'] = $phi->get_deleted(); $table['columns']['class'] = $phi->getClass(); */ //print_r($table); return $table; } /** * ORM * Map table data to phersistent instance */ public function data_to_phi($table) { $class = $this->full_class_name_to_simple_name($table['columns']['class']); $phi = $GLOBALS[$class]->create($table['columns']); return $phi; } public function get_table_name_ph(Phersistent $ph) { // table name declared in the class if (property_exists($ph, 'table')) { return $ph->table; } // go up in the inheritance until finding Phersistent while ($ph->get_parent() != 'phersistent\Phersistent') { $ph = $ph->get_parent_phersistent(); } $class_name = get_class($ph); return $this->class_to_table_name($class_name); } public function get_table_name(PhInstance $phi) { // TODO: should check STI and MTI (if part of STI, should return the name of the table where the STI is saved) // TODO: consider table name override declared on class // ** // For now inheritance ORM is all STI. // So need to check for parent = Phersistent, and that class will be the table name $ph = $phi->getDefinition(); return $this->get_table_name_ph($ph); } // For \a\b\TheClass, returns the_class // Convention: table names are snake case but class names are camel case private function class_to_table_name($class_name) { // \a\b\TheClass => TheClass $simple_name = $this->full_class_name_to_simple_name($class_name); // TheClass => the_class return

```
\basic\BasicString::camel_to_snake($simple_name); } /** * name of the column for the backlink
FK for one to many relationships. */ public function backlink_name($phi, $field) { $ph = $phi-
>getDefinition(); // table name declared in the class if (property_exists($ph, 'table')) { $prefix =
$ph->table; } else { // if CURRENT_CLASS(hasmany(assoc,OTHER_CLASS)) // then
$backlink_name = current_class_assoc_id // and that column should exist on the
OTHER_CLASS table $prefix = $this->class_to_table_name($phi->getClass()); } return
strtolower($prefix .'_'. $field .'_back'); } // Similar to previous method but $ph is Phersistent not
PhInstance. public function backlink_name_def($ph, $field) { if (property_exists($ph, 'table')) {
$prefix = $ph->table; } else { $prefix = $this->class_to_table_name(get_class($ph)); } return
strtolower($prefix .'_'. $field .'_back'); } /** * Maps each phersistent data type to a MySQL data
type. */ function get_db_type($phersistent_type) { // TODO: consider constraints like max_length
for texts switch ($phersistent_type) { case Phersistent::INT: return 'INT'; break; case
Phersistent::LONG: return 'BIGINT'; break; case Phersistent::FLOAT: return 'FLOAT'; break; case
Phersistent::DOUBLE: return 'DOUBLE'; break; case Phersistent::BOOLEAN: return 'BOOLEAN';
// synonym of TINYINT(1) break; case Phersistent::DATE: return 'DATE'; break; case
Phersistent::TIME: return 'TIME'; break; case Phersistent::DATETIME: return 'DATETIME'; break;
case Phersistent::DURATION: return 'INT'; // durations will be stored in seconds and converted
back to the duration expression // check https://stackoverflow.com/questions/13301142/php-how-
to-convert-string-duration-to-iso-8601-duration-format-ie-30-minute // check
https://gist.github.com/w0rldart/9e10aedd1ee55fc4bc74 break; case Phersistent::TEXT: return
'TEXT'; break; default: throw new \Exception('Data type '. $phersistent_type .' not supported'); } }
/** * $fullclass might include namespaces like \a\b\Class, this returns Class */ private function
full_class_name_to_simple_name($fullclass) { $parts = explode('\\', $fullclass); $class =
$parts[count($parts)-1]; return $class; } public function runRaw($sql) { $r = $this->driver-
>query($sql); // this checks for errors and throws exceptions $rows = array(); while ($row = $r-
>fetch_assoc()) { $rows[] = $row; } $r->close(); return $rows; } } ?>
```