Lesson Thursday

PHP (/php) / Database basics with PHP (/php/database-basics-with-php) / Summary

Text

This lesson is intended to supplement your notes from this week. **Do not copy and paste from it.** You will not learn anything and you will tie yourself in knots. Instead, use this as an example of one possible approach for how to start this type of problem from the ground up.

Create the database

- Plan out your relationships. A category has many tasks, and a task has one category. Or, a task has many categories and a category has one task.
- Create a project directory, and change directory into it in your terminal. Then, sign into the MySQL shell: `/Applications/MAMP/Library/bin/mysql --host=localhost -uroot -proot`
- Launch MAMP and start its server. Set its document root to your project folder in Preferences > Web Server by clicking the little folder icon.
- Now you can create your tables. Here are the most common SQL commands you will need.
- After creating your production database, copy it into your test database by going into phpMyAdmin, selecting the database, selecting the "Operations" tab, and running the copy operation.

```
CREATE DATABASE database_name;
USE database_name;
CREATE TABLE table_name (id serial PRIMARY KEY, some_column varchar (25
 another_column int, yet_another_column timestamp);
```

Other common commands: `SHOW DATABASES; SHOW TABLES; SELECT * FROM table_name; ALTER TABLE table_name ADD column_name boolean; ALTER TABLE table_name DROP column_name; INSERT INTO contacts (name, age, birthday) VALUES ('Wes', 43, '1969-05-01') RETURNING id; SELECT * FROM table_name WHERE age >= 18; UPDATE contacts SET name = 'Wes Anderson' WHERE id = 1; DELETE FROM contacts WHERE id = 1; DROP TABLE table_name; # From the database that holds the table. DROP DATABASE test_database; # From the $USER database.`

Naming conventions

- Remember that we want a production database and a test database following these naming conventions: `database_name` and `database_name_test`.
- Generally each table should have a column called `id` for the primary key.
- Generally tables should be in plural: `tasks` not `task`. We're storing a list of things afterall.
- If you are using a foreign key in your table, or using a join table for a many to many relationship you should name your columns `thing_id` as in `category_id` for tasks/categories. In the `categories` table we have a column for each category id called `id`, but when we are keeping track of which category a task belongs to, we call that column `category_id` to distinguish it from the task's `id` column.
- When naming a join table, use alphabetical order. For tasks and categories, we would generally have a table called `tasks`, a table called `categories` and a join table called `categories_tasks`.

Project Structure

Setup your Silex directories.

- `src` for your class files
- `tests` for your test files.
- `web` with `index.php`
- `app` with `app.php`
- `.gitignore` and `README.md`
- `views` folder for twig files.
- Create your `composer.json` file and install PHPUnit, Silex, Twig.
- Run `composer install`
- Don't start your Silex routes yet. Focus on tests first so that you know your methods work when you are building routes.

composer.json

```
{
    "require": {
        "phpunit/phpunit": "4.5.*",
        "silex/silex": "~1.1",
        "twig/twig":"~1.0"
    }
}
```

index.php

```
<?php
    $website = require_once __DIR__.'/../app/app.php';
    $website->run();
?>
```

app.php

```
<?php
    require_once __DIR__."/../vendor/autoload.php";
    require_once __DIR__."/../src/MyClass.php";

    $server = 'mysql:host=localhost:8889;dbname=to_do';
    $username = 'root';
    $password = 'root';
    $DB = new PDO($server, $username, $password);

    $app = new Silex\Application();

    $app->register(new Silex\Provider\TwigServiceProvider(), array(
        'twig.path' => __DIR__.'/../views'
    ));

    // routes

    return $app;
```

Don't forget to check your port number for localhost here by going into MAMP > Preferences > Ports > MySQL port.

TESTS

Again, focus on making your tests pass before writing Silex routes. Here is a skeleton test file:

```
<?php

    /**
    * @backupGlobals disabled
    * @backupStaticAttributes disabled
    */

    require_once "src/Task.php";

    $server = 'mysql:host=localhost:8889;dbname=to_do_test';
    $username = 'root';
    $password = 'root';
    $DB = new PDO($server, $username, $password);


    class TaskTest extends PHPUnit_Framework_TestCase
    {

        function test_name()
        {
            //Arrange

            //Act

            //Assert

        }
    }
?>
```

- Don't forget to check your localhost port number here too. Also check the name of your database. And don't forget the "backupGlobals" part. It looks like a comment but it isn't.

Getters and Setters

Start by building the structure of your objects. Write tests for your property getters and setters, making them pass one by one. Be sure to include a constructor with an id and make all your properties private.

If your objects have relationships, don't worry about them yet. Make sure that your objects have all the appropriate properties, as well as `save`, `getAll`, `deleteAll`, and `find` methods first. Always write a test for each method, then make it pass.

```
<?php

    /**
    * @backupGlobals disabled
    * @backupStaticAttributes disabled
    */

    require_once "src/Task.php";

    $server = 'mysql:host=localhost:8889;dbname=todo_test';
    $username = 'root';
    $password = 'root';
    $DB = new PDO($server, $username, $password);


    class TaskTest extends PHPUnit_Framework_TestCase
    {

        // Test your getters and setters.
        function test_getId()
        {
            //Arrange
            $id = 1;
            $description = "Watch the new Thor movie.";
            $test_task = new Task($description, $id);

            //Act
            $result = $test_task->getId();

            //Assert
            $this->assertEquals($id, $result); //make sure id returne
d is the one we put in, not null.
        }

        function test_getDescription()
        {
            //Arrange
            // no need to pass in id because it is null by default.
            $description = "Watch the new Thor movie.";
            $test_task = new Task($description);

            //Act
            $result = $test_task->getDescription();
            //Assert
            // id is null here, but that is not what we are testing.
 We are only interested in description.
            $this->assertEquals($description, $result);
        }

        function test_setDescription()
        {
            //Arrange
            $description = "Watch the new Thor movie.";
            $test_task = new Task($description);
            $new_description = "Watch the new Star Wars movie.";

            //Act
            $test_task->setDescription($new_description);
            $result = $test_task->getDescription();

            //Assert
            $this->assertEquals($new_description, $result);
        }
    }
?>
```

```php
<?php
class Task
{
    private $description;
    private $id;

    function __construct($description, $id = null)
    {
        $this->description = $description;
        $this->id = $id;
    }

    function getId()
    {
        return $this->id;
    }

    function getDescription()
    {
        return $this->description;
    }

    function setDescription($new_description)
    {
        $this->description = (string) $new_description;
    }
}
?>
```

Repeat this step for your other models.

Save, GetAll, DeleteAll

- These methods need to be built together because their tests rely on each other. **But write the tests first, and then make them pass, one class at a time.**
- Don't forget to add your teardown function, and when you add a second model clear out its table in your teardown function too.

```
        protected function tearDown()
        {
          Task::deleteAll();
          Category::deleteAll();
        }

        function test_save()
        {
            //Arrange
            $description = "Eat breakfast.";
            $test_task = new Task($description);
            $test_task->save(); // id gets created by database and wr
itten in during save method.

            //Act
            $result = Task::getAll();

            //Assert
            $this->assertEquals($test_task, $result[0]);
        }

        function test_getAll()
        {
            //Arrange
            // create more than one task to make sure getAll returns
 them all.
            $description = "Eat breakfast.";
            $description2 = "Eat lunch.";
            $test_task = new Task($description);
            $test_task->save();
            $test_task2 = new Task($description2);
            $test_task2->save();

            //Act
            $result = Task::getAll();

            //Assert
            $this->assertEquals([$test_task, $test_task2], $result);
        }

        function test_deleteAll()
        {
            //Arrange
            // create more than one task
            $description = "Eat breakfast.";
            $description2 = "Eat lunch.";
            $test_task = new Task($description);
            $test_task->save();
            $test_task2 = new Task($description2);
            $test_task2->save();

            //Act
            Task::deleteAll(); // delete them.
            $result = Task::getAll(); // get all to make sure they ar
e gone.

            //Assert
            $this->assertEquals([], $result);
        }
```

```
<?php
class Task
{
    private $description;
    private $id;

    function __construct($description, $id = null)
    {
        $this->description = $description;
        $this->id = $id;
    }

    function getId()
    {
        return $this->id;
    }

    function getDescription()
    {
        return $this->description;
    }

    function setDescription($new_description)
    {
        $this->description = (string) $new_description;
    }

    function save()
    {
        $GLOBALS['DB']->exec("INSERT INTO tasks (description) VALUES
 ('{$this->getDescription()}')");
        $this->id = $GLOBALS['DB']->lastInsertId();
    }

    static function getAll()
    {
        $returned_tasks = $GLOBALS['DB']->query("SELECT * FROM task
s;");
        $tasks = array();
        foreach($returned_tasks as $task) {
            $description = $task['description'];
            $id = $task['id'];
            $new_task = new Task($description, $id);
            array_push($tasks, $new_task);
        }
        return $tasks;
    }

    static function deleteAll()
    {
      $GLOBALS['DB']->exec("DELETE FROM tasks;");
    }
}
?>
```

Find method

Next, we need to be able to locate an individual instance of a class by its id number. We'll write the test, and then add the method. Repeat for both classes.

```
        function test_find()
        {
            //Arrange
            // create more than one task so that we can be sure we ca
n locate the one we are interested in.
            $description = "Eat breakfast.";
            $description2 = "Eat lunch.";
            $test_task = new Task($description);
            $test_task->save(); // id is assigned to task by databas
e.
            $test_task2 = new Task($description2);
            $test_task2->save();

            //Act
            // find a task by using the id assigned during the save m
ethod.
            $result = Task::find($test_task->getId());

            //Assert
            // make sure we found the one we were looking for.
            $this->assertEquals($test_task, $result);
        }
```

```
    static function find($search_id)
    {
        $found_task = null;
        $tasks = Task::getAll();
        foreach($tasks as $task) {
            $task_id = $task->getId();
            if ($task_id == $search_id) {
              $found_task = $task;
            }
        }
        return $found_task;
    }
```

Add relationships between models

- For a one-to-many relationship, the id of one class should be kept track of in the other class's table. For example, if a task has one category, but a category has many tasks, it makes sense to have a `category_id` column in your tasks table. This way we can assign a task to a category by id, and we can get all the tasks assigned to one category by searching through the tasks table for that category id.
- For a many-to-many relationship, we need a join table where the ids from each class are associated with eachother by adding rows to the join table. For example, if a task has many categories, but a category also has many tasks, we need a join table: `categories_tasks` . Then each row in that table would tie a `task_id` to a `category_id` . Each row would have its own primary key id, essentially its row number, but it would also have two columns for the ids of each class.

Adding a one-to-many relationship

- We're going to add a category_id to our tasks table because a category has many tasks, but a task has only one category.
- Alter your table to add the new column: `ALTER TABLE tasks ADD category_id int;`
- Drop your test database and make a new copy of your production database so that your test database matches your production database with the new column.
- Now we have to alter all of our tests to include the new column. **Since this has to be a real id that actually exists in both tables, we have to expand our arrange steps to include creating instances of both models. We**

can't assign a task to a category that doesn't exist, we have to actually make categories too.

- Since we will be adding a property to our task object, it will need a getter and a setter, so we will want tests for those methods.

```
<?php

    /**
    * @backupGlobals disabled
    * @backupStaticAttributes disabled
    */

    require_once "src/Task.php";
    require_once "src/Category.php";

    $server = 'mysql:host=localhost:8889;dbname=todo_test';
    $username = 'root';
    $password = 'root';
    $DB = new PDO($server, $username, $password);


    class TaskTest extends PHPUnit_Framework_TestCase
    {
        protected function tearDown()
        {
          Category::deleteAll();
          Task::deleteAll();
        }

        // Test your getters and setters.
        function test_getId()
        {
            //Arrange
            // create new category and save it so that it has an id.
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();

            $id = 1;
            $description = "Watch the new Thor movie.";
            // add the category id as the second argument instead of
 the last one because it isn't optional.
            $test_task = new Task($description, $category_id, $id);

            //Act
            $result = $test_task->getId();

            //Assert
            $this->assertEquals($id, $result); //make sure id returne
d is the one we put in, not null.
        }

        function test_getDescription()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();
            // no need to pass in id because it is null by default.
            $description = "Watch the new Thor movie.";
            $test_task = new Task($description, $category_id);

            //Act
            $result = $test_task->getDescription();

            //Assert
            // id is null here, but that is not what we are testing.
 We are only interested in description.
            $this->assertEquals($description, $result);
        }

        function test_setDescription()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();

            $description = "Watch the new Thor movie.";
            $test_task = new Task($description, $category_id);
            $new_description = "Watch the new Star Wars movie.";

            //Act
            $test_task->setDescription($new_description);
            $result = $test_task->getDescription();

            //Assert
            $this->assertEquals($new_description, $result);
        }
        function test_getCategoryId()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();
            // no need to pass in id because it is null by default.
            $description = "Watch the new Thor movie.";
            $test_task = new Task($description, $category_id);

            //Act
            $result = $test_task->getCategoryId();

            //Assert
            // id is null here, but that is not what we are testing.
 We are only interested in description.
            $this->assertEquals($category_id, $result);
        }

        function test_setCategoryId()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();
            // create a second category so that we can change the cat
egory our task is assigned to.
            $name = "Books to Read";
            $test_category2 = new Category($name);
            $test_category2->save();
            $category_id2 = $test_category2->getId();

            $description = "Watch the new Thor movie.";
            $test_task = new Task($description, $category_id);

            //Act
            // assign task to a new category
            $test_task->setCategoryId($category_id2);
            $result = $test_task->getCategoryId();

            //Assert
            // make sure that the task's category id has changed.
            $this->assertEquals($category_id2, $result);
        }

        function test_save()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();

            $description = "Eat breakfast.";
            $test_task = new Task($description, $category_id);
            $test_task->save(); // id gets created by database and wr
itten in during save method.

            //Act
            $result = Task::getAll();

            //Assert
            $this->assertEquals($test_task, $result[0]);
        }

        function test_getAll()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();

            // create more than one task to make sure getAll returns
 them all.
            $description = "Eat breakfast.";
            $description2 = "Eat lunch.";
            $test_task = new Task($description, $category_id);
            $test_task->save();
            $test_task2 = new Task($description2, $category_id);
            $test_task2->save();

            //Act
            $result = Task::getAll();

            //Assert
            $this->assertEquals([$test_task, $test_task2], $result);
        }

        function test_deleteAll()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();

            // create more than one task
            $description = "Eat breakfast.";
            $description2 = "Eat lunch.";
            $test_task = new Task($description, $category_id);
            $test_task->save();
            $test_task2 = new Task($description2, $category_id);
            $test_task2->save();

            //Act
            Task::deleteAll(); // delete them.
            $result = Task::getAll(); // get all to make sure they ar
e gone.

            //Assert
            $this->assertEquals([], $result);
        }

        function test_find()
        {
            //Arrange
            $name = "Movies To Watch";
            $test_category = new Category($name);
            $test_category->save();
            $category_id = $test_category->getId();

            // create more than one task so that we can be sure we ca
n locate the one we are interested in.
            $description = "Eat breakfast.";
            $description2 = "Eat lunch.";
            $test_task = new Task($description, $category_id);
            $test_task->save(); // id is assigned to task by databas
e.
            $test_task2 = new Task($description2, $category_id);
            $test_task2->save();

            //Act
            // find a task by using the id assigned during the save m
ethod.
            $result = Task::find($test_task->getId());

            //Assert
            // make sure we found the one we were looking for.
            $this->assertEquals($test_task, $result);
        }
    }
?>
```

Categories have not changed, so there's no need to change those tests. Then, we need to modify our Task.

- We need a property to store the category id, and we need to input it with the constructor. It will also need a getter and a setter.
- We need to alter the save method to write the category id into the database. **We can't create new records in the database to save our objects if we don't write data into every column.** Remember, it's an integer and not a string so we don't want quotes around the the id property in our insert statement.
- We need to alter the getAll method to read the category id out of the database.
- Delete and find methods do not depend on category ids, so nothing needs to change there.

```
<?php
class Task
{
    private $description;
    private $id;
    private $category_id;

    function __construct($description, $assigned_category_id, $id = n
ull)
    {
        $this->description = $description;
        $this->id = $id;
        $this->category_id = $assigned_category_id;
    }

    function getId()
    {
        return $this->id;
    }

    function getDescription()
    {
        return $this->description;
    }

    function getCategoryId()
    {
        return $this->category_id;
    }

    function setDescription($new_description)
    {
        $this->description = (string) $new_description;
    }

    function setCategoryId($new_category_id)
    {
        $this->category_id = (int) $new_category_id;
    }

    function save()
    {
        $GLOBALS['DB']->exec("INSERT INTO tasks (description, categor
y_id) VALUES ('{$this->getDescription()}', {$this-
>getCategoryId()})");
        $this->id = $GLOBALS['DB']->lastInsertId();
    }

    static function getAll()
    {
        $returned_tasks = $GLOBALS['DB']->query("SELECT * FROM task
s;");
        $tasks = array();
        foreach($returned_tasks as $task) {
            $description = $task['description'];
            $id = $task['id'];
            $category_id = $task['category_id'];
            $new_task = new Task($description, $category_id, $id);
            array_push($tasks, $new_task);
        }
        return $tasks;
    }

    static function deleteAll()
    {
      $GLOBALS['DB']->exec("DELETE FROM tasks;");
    }

    static function find($search_id)
    {
        $found_task = null;
        $tasks = Task::getAll();
        foreach($tasks as $task) {
            $task_id = $task->getId();
            if ($task_id == $search_id) {
              $found_task = $task;
            }
        }
        return $found_task;
    }
}
?>
```

Find all the tasks that belong to a category.

The advantage of having a one-to-many relationship is that we can do more advanced searches. Let's create a test for finding all the tasks assigned to one category id. Since the piece of information we have relates to categories, we want to put this test in our CategoryTest.php file. We are saying "Hey category, give me all the tasks that are assigned to your id."

```
        function testGetTasks()
        {
            //Arrange
            // create two categories.
            $name = "Movies To Watch";
            $name2 = "Books To Read";
            $test_category = new Category($name);
            $test_category->save(); // id is assigned to category by
 database.
            $test_category_id = $test_category->getId();
            $test_category2 = new Category($name2);
            $test_category2->save();
            $test_category_id2 = $test_category2->getId();

            // create three tasks. Assign them to different categorie
s.
            $description = "Lord of the Rings";
            $test_task = new Task($description, $test_category_id);
            $test_task->save();

            $description2 = "Hackers";
            $test_task2 = new Task($description2, $test_category_id);
            $test_task2->save();

            // assign this task to category2 so that we get category
1's tasks, they are the correct ones.
            $description3 = "Neuromancer";
            $test_task3 = new Task($description3,
$test_category_id2);
            $test_task3->save();

            //Act
            // our getTasks() method should return us an array of tas
ks, similarly to our getAll method.
            // but instead of returning all tasks stored in the datab
ase, we only want the ones associated with the given category id.
            $result = $test_category->getTasks();

            //Assert
            // be sure that our result does not include $test_task3.
            $this->assertEquals([$test_task, $test_task2], $result);
        }

    }
    function getTasks()
    {
        $tasks = Array();
        $returned_tasks = $GLOBALS['DB']->query("SELECT * FROM tasks
WHERE category_id = {$this->getId()};");
        foreach($returned_tasks as $task) {
            $description = $task['description'];
            $id = $task['id'];
            $category_id = $task['category_id'];
            $new_task = new Task($description, $category_id, $id);
            array_push($tasks, $new_task);
        }
        return $tasks;
    }
```

Add The User Interface

Now that we are confident that our main backend methods are working because all our tests are passing, it is safe to focus on our frontend for awhile. See these lessons for a guide on how to do that:

- https://www.learnhowtoprogram.com/php/database-basics-with-php/to-do-with-sql-and-silex
- https://www.learnhowtoprogram.com/php/database-basics-with-php/relationships-in-silex

Add the Edit and Delete functionality

In order to have full CRUD functionality, (Create, Read, Update, Destroy) we now need to add Update and Destroy methods for editing or deleting singular objects. This will involve writing two new methods, so we will need to write two new tests first. Since we are going to be editing/deleting one object at a time, we'll need to use our `find` method to get the correct object record out of the database. Then, we can update our user interface.

Follow along with this lesson for a guide on edit/delete.

- https://www.learnhowtoprogram.com/php/database-basics-with-php/using-patch-and-delete-in-php

Export your database

Don't forget to save your work by exporting your database at the end, and include information about your database with all setup instructions in your readme.

- https://www.learnhowtoprogram.com/php/database-basics-with-php/exporting-mysql-databases-in-phpmyadmin