Building a Website with PHP, MySQL and jQuery Mobile, Part 1

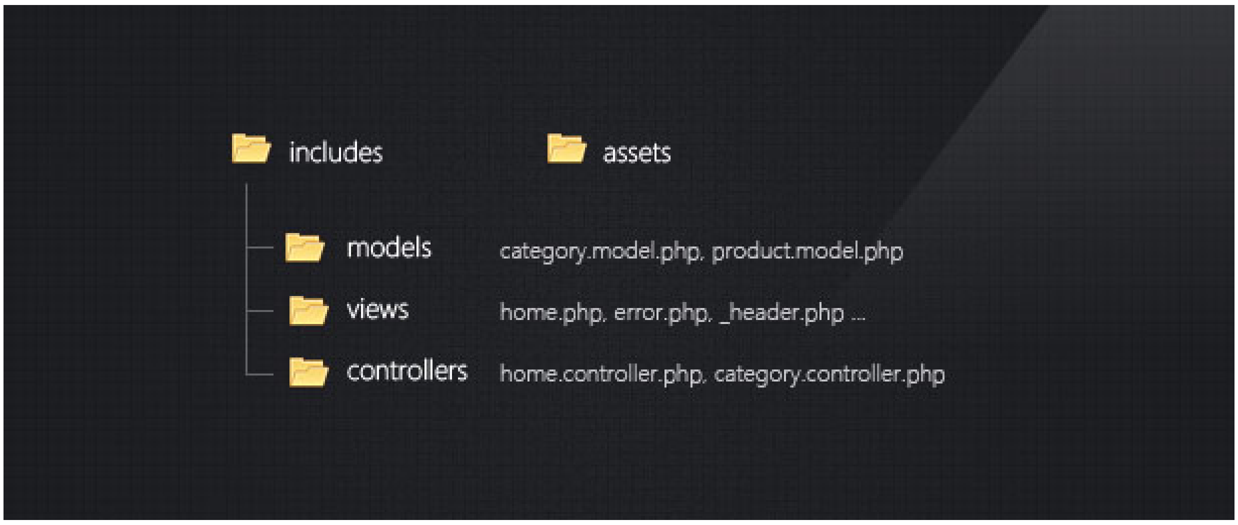
Martin Angelov August 19th, 2011

jQuery Trickshots is our new epic jQuery tips and tricks book. Check it out! [1]
In this two-part tutorial, we will be building a simple website with PHP and MySQL, using the Model-View-Controller (MVC) pattern. Finally, with the help of the jQuery Mobile framework [2], we will turn it into a touch-friendly mobile website, that works on any device and screen size.

In this first part, we concentrate on the backend, discussing the database and MVC organization. In part two [3], we are writing the views and integrating jQuery Mobile.

The File Structure

As we will be implementing the MVC pattern (in effect writing a simple micro-framework), it is natural to split our site structure into different folders for the models, views and controllers. Don’t let the number of files scare you – although we are using a lot of files, the code is concise and easy to follow.

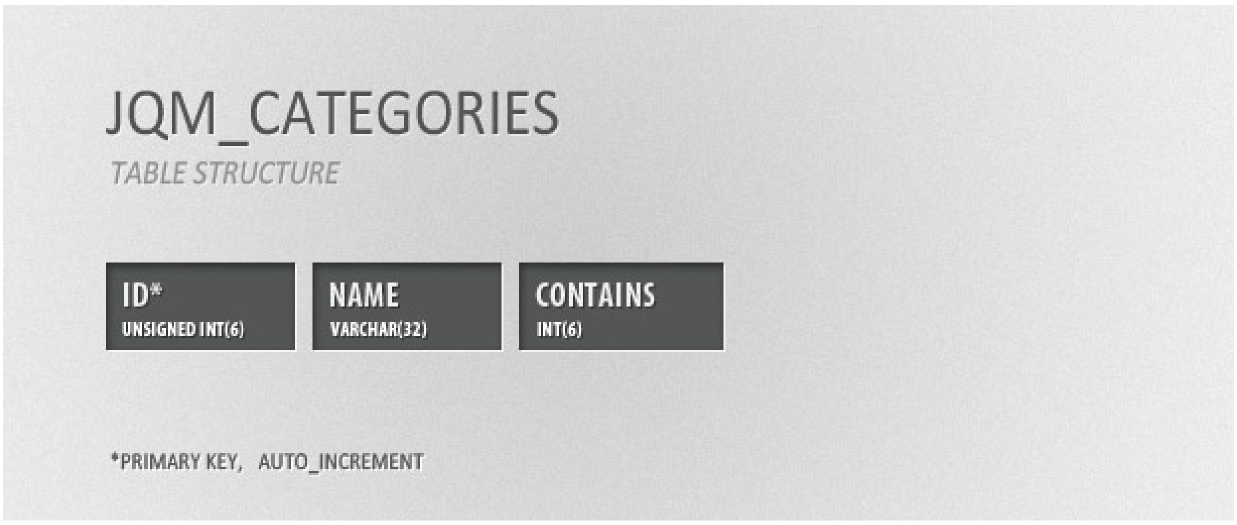

includes
assets
models
category.model.php, product.model.php
views
home.php, error.php, _header.php ...
controllers
home.controller.php, category.controller.php

[4]

The Directory Structure

The Database Schema

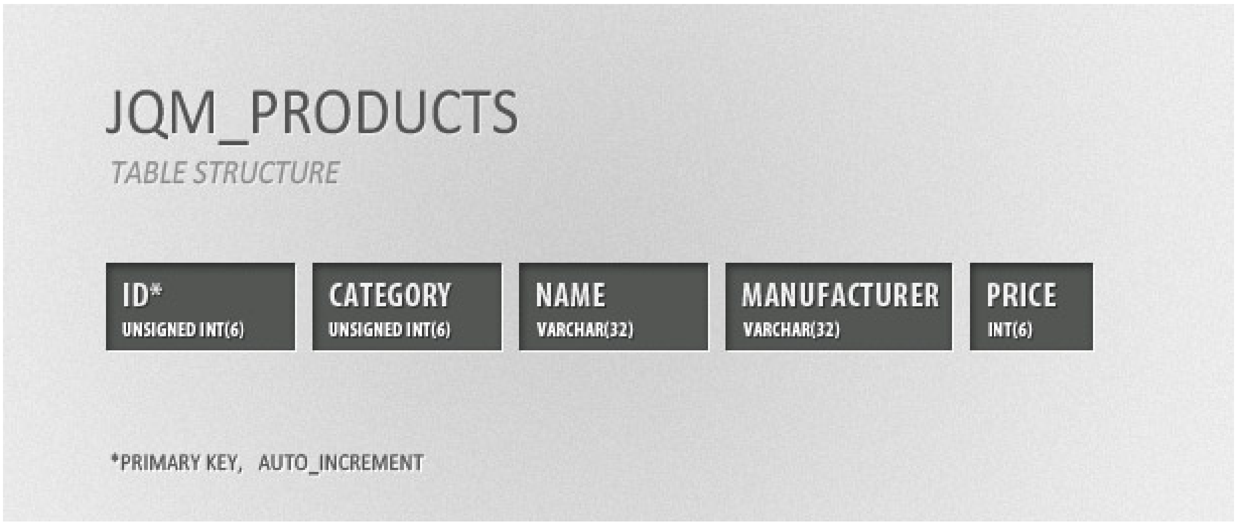
Our simple application operates with two types of resources – categories and products. These are given their own tables – **jqm_categories**, and **jqm_products**. Each product has a category field, which assigns it to a category.



[5]

jqm_categories Table Structure

The categories table has an **ID** field, a **name** and a **contains** column, which shows how many products there are in each category.



[6]

jqm_products Table Structure

The product table has a **name**, **manufacturer**, **price** and a **category** field. The latter holds the ID of the category the product is added to.

> *You can find the SQL code to create these tables in tables.sql in the download archive. Execute it in the SQL tab of phpMyAdmin to have a*

> *working copy of this database. Remember to also fill in your MySQL login details in config.php.*

The Models

The models in our application will handle the communication with the database. We have two types of resources in our application – **products** and **categories**. The models will expose an easy to use method – `find()` which will query the database behind the scenes and return an array with objects.

Before starting work on the models, we will need to establish a database connection. I am using the PHP PDO class [7], which means that it would be easy to use a different database than MySQL, if you need to.

includes/connect.php

```
/*
   This file creates a new MySQL connection using the PDO class.
   The login details are taken from includes/config.php.
*/

try {
   $db = new PDO(
      "mysql:host=$db_host;dbname=$db_name;charset=UTF-8",
      $db_user,
      $db_pass
   );

   $db->query("SET NAMES 'utf8'");
   $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch(PDOException $e) {
   error_log($e->getMessage());
   die("A database error was encountered");
}
```

This will put the **$db** connection object in the global scope, which we will use in our models. You can see them below.

includes/models/category.model.php

```
class Category{

   /*
      The find static method selects categories
      from the database and returns them as
      an array of Category objects.
   */

   public static function find($arr = array()){
      global $db;

      if(empty($arr)){
         $st = $db->prepare("SELECT * FROM jqm_categories");
      }
      else if($arr['id']){
         $st = $db->prepare("SELECT * FROM jqm_categories WHERE id=:id");
      }
      else{
         throw new Exception("Unsupported property!");
      }

            // This will execute the query, binding the $arr values as query parameters
      $st->execute($arr);

      // Returns an array of Category objects:
      return $st->fetchAll(PDO::FETCH_CLASS, "Category");
   }
}
```

Both models are simple class definitions with a single static method – **find()**. In the fragment above, this method takes an optional array as a parameter and executes different queries as prepared statements.

In the return declaration, we are using the fetchAll [8] method passing it the **PDO::FETCH_CLASS** constant. What this does, is to loop though all the result rows, and create a new object of the Category class. The columns of each row will be added as public properties to the object.

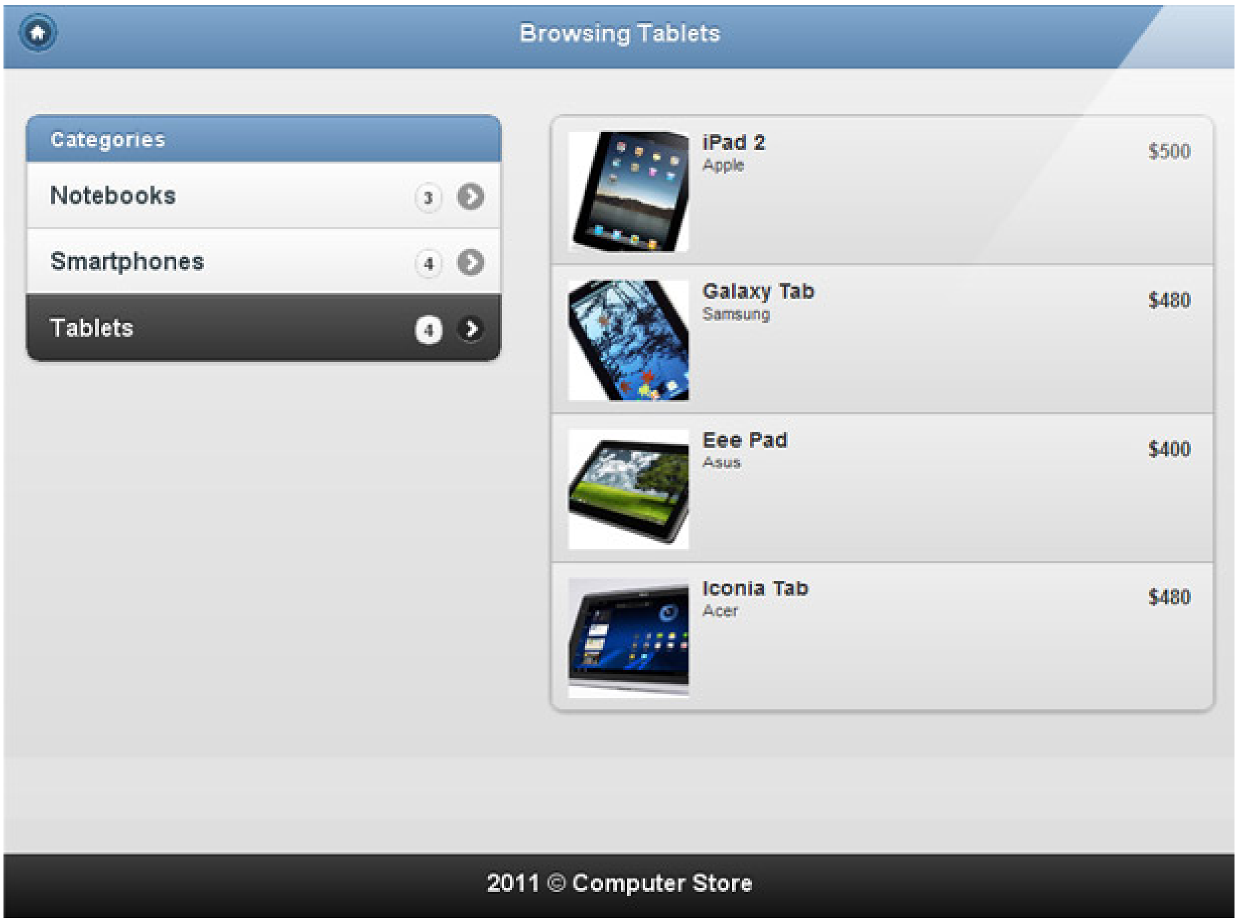
This is also the case with the *Product model*:

includes/models/product.model.php

```
class Product{

   // The find static method returns an array
   // with Product objects from the database.

   public static function find($arr){
      global $db;

      if($arr['id']){
         $st = $db->prepare("SELECT * FROM jqm_products WHERE id=:id");
      }
      else if($arr['category']){
         $st = $db->prepare("SELECT * FROM jqm_products WHERE category = :category");
      }
      else{
         throw new Exception("Unsupported property!");
      }

      $st->execute($arr);

      return $st->fetchAll(PDO::FETCH_CLASS, "Product");
   }
}
```

The return values of both find methods are arrays with instances of the class. We could possibly return an array of generic objects (or an array of arrays) in the find method, but creating specific instances will allow us to automatically style each object using the appropriate template in the views folder (the ones that start with an underscore). We will talk again about this in the next part of the tutorial.

There, now that we have our two models, lets move on with the controllers.



[9]

Computer Store with PHP, MySQL and jQuery Mobile

The controllers

The controllers use the **find()** methods of the models to fetch data, and render the appropriate views. We have two controllers in our application – one for the *home page*, and another one for the *category pages*.

includes/controllers/home.controller.php

```
/* This controller renders the home page */

class HomeController{
    public function handleRequest(){
        // Select all the categories:
        $content = Category::find();

        render('home',array(
            'title'         => 'Welcome to our computer store',
            'content'       => $content
        ));
    }
}
```

Each controller defines a **handleRequest()** method. This method is called when a specific URL is visited. We will return to this in a second, when we discuss *index.php*.

In the case with the **HomeController**, **handleRequest()** just selects all the categories using the model's find() method, and renders the home view (*includes/views/home.php*) using our *render* helper function (*includes/helpers.php*), passing a title and the selected categories. Things are a bit more complex in **CategoryController**:

includes/controllers/category.controller.php

```
/* This controller renders the category pages */

class CategoryController{
    public function handleRequest(){
        $cat = Category::find(array('id'=>$_GET['category']));

        if(empty($cat)){
            throw new Exception("There is no such category!");
        }

        // Fetch all the categories:
        $categories = Category::find();

        // Fetch all the products in this category:
        $products = Product::find(array('category'=>$_GET['category']));
        // $categories and $products are both arrays with objects

        render('category',array(
            'title'         => 'Browsing '.$cat[0]->name,
            'categories'    => $categories,
            'products'      => $products
        ));
    }
}
```

The first thing this controller does, is to select the category by id (it is passed as part of the URL). If everything goes to plan, it fetches a list of categories, and a list of products associated with the current one. Finally, the category view is rendered.

Now lets see how all of these work together, by inspecting *index.php*:

index.php

```
/*
    This is the index file of our simple website.
    It routes requests to the appropriate controllers
*/

require_once "includes/main.php";

try {

    if($_GET['category']){
        $c = new CategoryController();
    }
    else if(empty($_GET)){
        $c = new HomeController();
    }
    else throw new Exception('Wrong page!');

    $c->handleRequest();
}
catch(Exception $e) {
    // Display the error page using the "render()" helper function:
    render('error',array('message'=>$e->getMessage()));
}
```

This is the first file that is called on a new request. Depending on the **$_GET** parameters, it creates a new controller object and executes its **handleRequest()** method. If something goes wrong anywhere in the application, an exception will be generated which will find its way to the catch clause, and then in the error template.

One more thing that is worth noting, is the very first line of this file, where we require *main.php*. You can see it below:

main.php

```
/*
    This is the main include file.
    It is only used in index.php and keeps it much cleaner.
*/

require_once "includes/config.php";
require_once "includes/connect.php";
require_once "includes/helpers.php";
require_once "includes/models/product.model.php";
require_once "includes/models/category.model.php";
require_once "includes/controllers/home.controller.php";
require_once "includes/controllers/category.controller.php";

// This will allow the browser to cache the pages of the store.

header('Cache-Control: max-age=3600, public');
header('Pragma: cache');
header("Last-Modified: ".gmdate("D, d M Y H:i:s",time())." GMT");
header("Expires: ".gmdate("D, d M Y H:i:s",time()+3600)." GMT");
```

This file holds the **require_once** declarations for all the models, controllers and helper files. It also defines a few headers to enable caching in the browser (PHP disables caching by default), which speeds up the performance of the jQuery mobile framework.

Continue to Part 2

With this the first part of the tutorial is complete! Continue to part 2 [10], where we will be writing the views and incorporate jQuery Mobile. Feel free to share your thoughts and suggestions in the comment section below.

by Martin Angelov

Martin is a web developer with an eye for design from Bulgaria. He founded Tutorialzine in 2009 and publishes new tutorials weekly.

Tutorials [11]

1. http://tutorialzine.com/books/jquery-trickshots/
2. http://jquerymobile.com/
3. http://tutorialzine.com/2011/08/jquery-mobile-mvc-website-part-2/
4. http://demo.tutorialzine.com/2011/08/jquery-mobile-product-website/
5. http://demo.tutorialzine.com/2011/08/jquery-mobile-product-website/
6. http://demo.tutorialzine.com/2011/08/jquery-mobile-product-website/
7. http://www.php.net/manual/en/intro.pdo.php
8. http://www.php.net/manual/en/pdostatement.fetchall.php
9. http://demo.tutorialzine.com/2011/08/jquery-mobile-product-website/
10. http://tutorialzine.com/2011/08/jquery-mobile-mvc-website-part-2/
11. http://tutorialzine.com/category/tutorials/