Building Dynamic Websites With the MVC Pattern

ACM Webmonkeys @ UIUC, 2010

Recap

- A **dynamic** website is a website which uses some server-side language to **generate** HTML pages
- PHP is a common and ubiquitous server-side language, but there are many options
- Imagine if you wrote a program in Java which, instead of outputting unformatted text to the console, output HTML
  - Any language at all can be used to generate static pages, so long as the web server is configured right
  - Parts of some big websites (e.g. Google) even use C++ for speed-critical tasks

Revisiting PHP: Classes

- A **class** is a definition of an **object**, which is a data structure with associated properties and methods.
    - A class must be **instantiated** into an object
- Classes in PHP work very similar to how classes work in C++ or Java.

```
class MyClass {
    var $myvar; // Remember, variables are dynamically-typed.

    function doSomething($str) {
        $this->myvar = $str;
    }
}
```

Take note that $this is the reference for the object which doSomething was called on -- the arrow syntax (->) is how one accesses a method or property of a class.

Working with objects

- To instantiate an object, use the **new** operator:
  - $myobj = new MyClass();
- To call a method (or access a variable) of a class, use an arrow (->).
  - $myobj->doSomething("blah");
  - $var = $myobj->myvar;
    - Note how the $ is **not** repeated.
- You don't need to explicitly delete objects
  - PHP has automated garbage collection

Why bother with classes?

- Classes provide a way to organize your code into separate pieces.
  - You can avoid duplicating code by inheriting methods from superclasses
  - Wrap functional parts of your code into "black boxes" -- that is, provide an interface for other code to use but don't concern them with the details
- But how does this apply to websites?
  - Key point: programming a large website is not very different from programming a large desktop application. How can we apply the techniques we learn in CS to websites?

First, analyze what a website does

- A non-trivial dynamic website will probably, upon receiving a request from the user for a page, do the following:
  1. Determine **what** the user wants to do (view a list of products, view product details, buy a product, etc.)
  2. **Find** the data relevant to the task (from a database!)
  3. Potentially manipulate or **modify** the data
  4. **Display** a result to the user

The direct approach

- The direct approach is to create a different page for each task, and implement those four steps on each. So we might have:
  - welcome.php
  - product_listing.php
  - product_details.php
  - purchase.php
- This lets us know *what* the user wants to do fairly easily (just by virtue of which page we're on).
- However, there's still a lot of common code between each page. For example, the outer frame of the site's layout isn't changing.

DRY: Don't Repeat Yourself

- A term used frequently in the Ruby on Rails community is **DRY**: Don't Repeat Yourself.
  - In other words, eliminate redundancies from your code.
- If we have the same header and footer on every page of our website, what if we need to make a minor change?
  - Need to change every single file in the same way!
  - A real headache to maintain.
- If we want to avoid repeating ourselves, we should put all the common code for the header and footer in one place, like a PHP function, and simply call that function on each page.

Factoring out the header, method 1:

```
<?php
// global.php
//  A library of common
//  functions for the
//  site.

function header() {
    echo "<html>....";
    // etc, etc.
}

function footer() {
    echo "....</html>";
}

?>
```

```
<?php
// welcome.php
//  The homepage.
include('global.php');

header();

echo "Welcome to
our site!";

footer();
?>
```

Pretty clean, eh?

What about OOP?

- That approach works fine, but it lacks the logical separation that OOP can give us.
  - Not that the previous approach won't work, but for larger sites, it can get messy.
- What if we wanted to turn each page into a class, all of which extend from a superclass?
  - The superclass can worry about how to print the header and footer.
  - All each class has to do is override the printPageContent() method (for example).

So we define a superclass:

```
class Page {
  function header() {
    echo "<html><body><h1>My Site</h1>";
  }
  function footer() {
    echo "</body></html>";
  }
  function printPageContent() {
    /* Needs to be defined in subclasses. */
  }

  function render() {
     $this->header();
     $this->printPageContent();
     $this->footer();
  }
}
```

And each separate page looks like:

```
class WelcomePage extends Page {
  function printPageContent() {
    echo "<b>Welcome to my site!</b>";
  }
}
```

- That's it -- the other functions are already defined in the Page superclass.
- To print this page, we do:
  - $page = new WelcomePage();
  - $page->render();

So what's the point?

- The key idea is that we want to separate parts of the application into logical components, and minimize redundancies.
- In this simple context, it's largely unnecessary. For larger sites, it becomes more useful:
    - Being able to render a page at-will (just by instantiating it and calling the render() method) lets us dynamically route URLs to different pages
    - If two pages are similar, we can extend one from the other and reduce redundancy
        - That is, we can apply OOP principles to web pages

Getting dynamic

- So far, what we've done isn't really any better than just writing static HTML files.
- For our example, we're considering a store's website, so how do we make it dynamic?
  - Directly listing products could be done in static HTML as well.
  - We can make it dynamic by storing the list of products in a **database**, and retrieving it on each page load.
    - Saves time when maintaining the list of products
- In general, dynamic (changing) content is loaded from a database.
  - Not always, but it's a very common case.

Pulling data from a database

- As we went over briefly last week, pulling data from a database is done by first running an SQL 'SELECT' query on the database, then iterating over the results returned.
    - $qry = mysql_query("SELECT * FROM products;");
    - while ($row = mysql_fetch_assoc($qry)) {
    -     echo $row['name'] . ", " . $row['price'];
    - }
- The downside of this approach is that it relies heavily on:
    - always using MySQL
    - not changing your database schema
- What if we change the name of a table somewhere?
    - potentially hundreds of queries to modify

Can we abstract database querying?

- Not easily: a database is simply a collection of tables, which in turn are a collection of data items.
  - Unlike with pages, we cannot easily use OOP principles to clean up our database code
- What we can do, however, is **encapsulate** the parts of our code that deal with the database
  - That is, write functions like getAllProducts(), getProductDetails($id), etc.
  - This way, if we later change how the query for each function is written, we only need to change it in one place.

Taking it a step further...

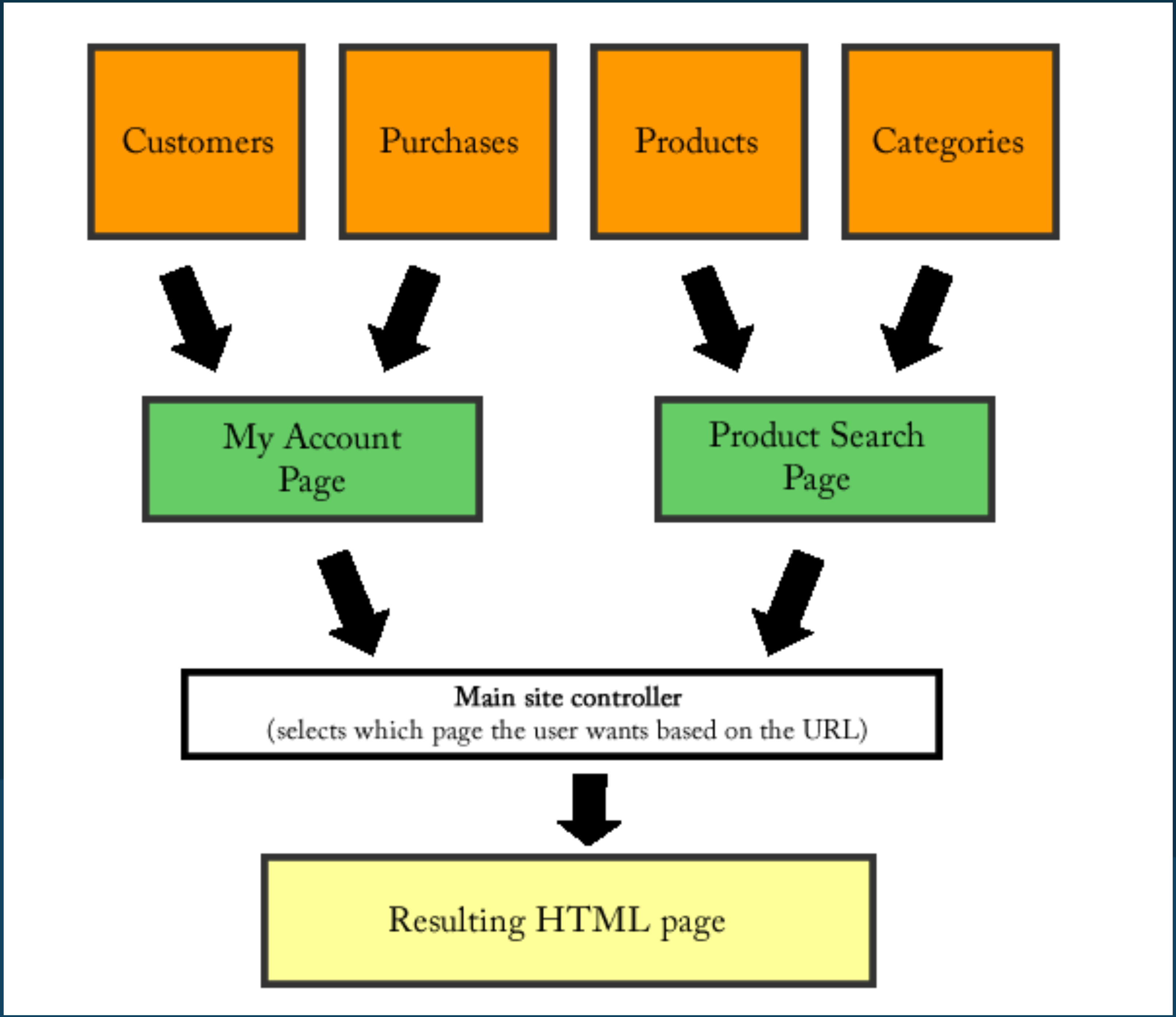
```
// in product.php:
class Product {
  function getAllProducts() {
    $products = array();

    $qry = mysql_query("SELECT * FROM products;");
    while ($product = mysql_fetch_assoc($qry)) {
      $products[] = $product;
    }
    return $products;
  }
}

// in product_listing.php:
...

$products = Product::getAllProducts();

foreach ($products as $product) {
  echo "<tr><td>" . $product['name'] . "</td><td>Buy 
Me</td></tr>";
}
...
```

What have we done?

- We created a Product class, here mainly just to contain our product-related functions, and put our get-product-data-from-the-database functions into it as static methods.
- We then used the function from the Product class in our actual page
  - This means callers don't have to care where the product data comes from
- We could go further...
  - Instead of returning an array of flat product data, what if we added properties to Product and returned an array of instances of Product?

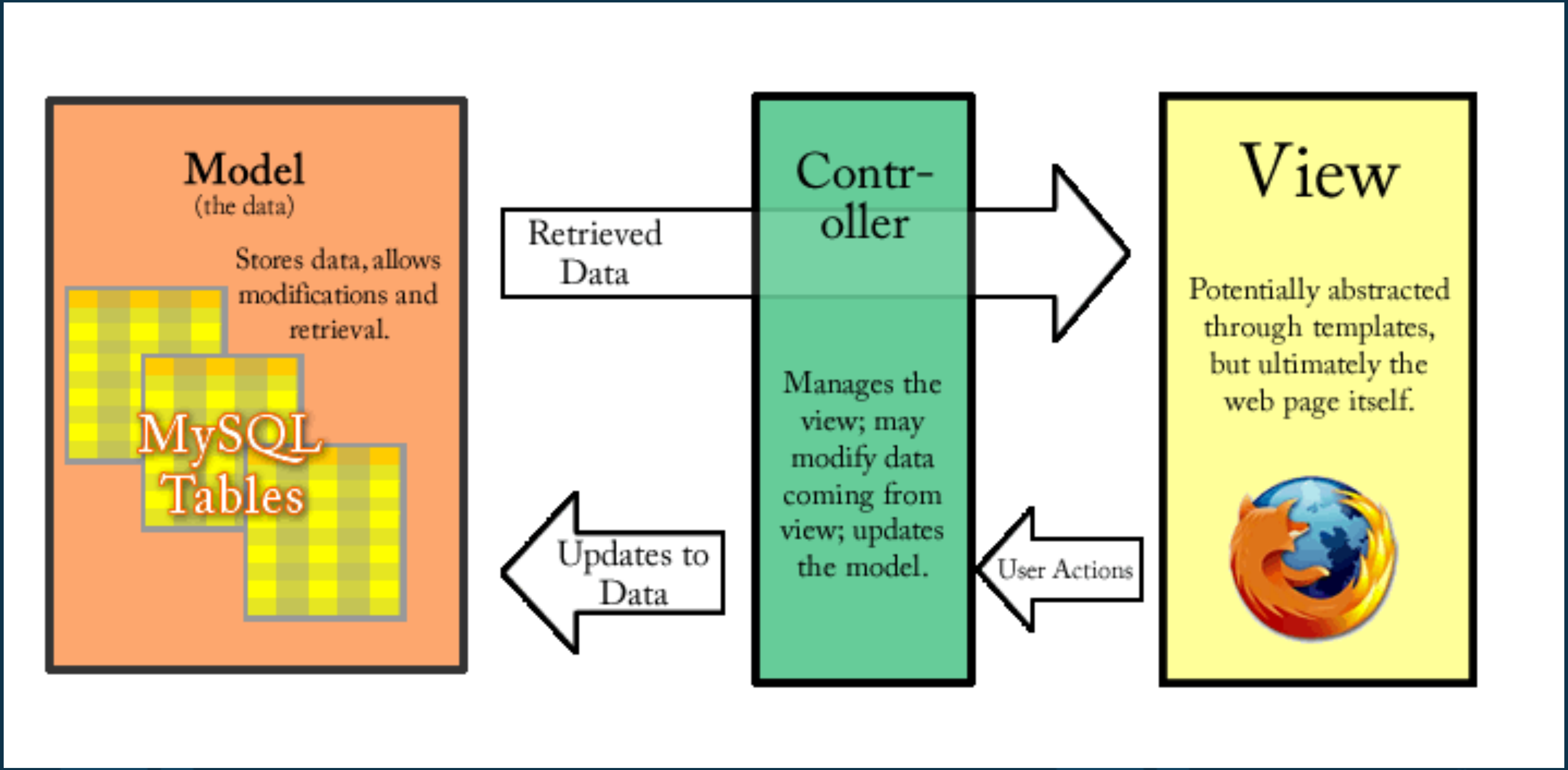
Take a step back

- So, if we were to copy this pattern across every data type we have in our database, we would have all our database code abstracted.
- Then, we have Page classes which use functions from the Data classes to display pages.
- The ultimate result gets printed out to HTML and sent back to the user.
- Key points:
    - The Page classes do not worry about the details of the database.
    - The HTML, of course, does not worry at all about the database either.


Customers
Purchases
Products
Categories
My Account Page
Product Search Page
Main site controller
(selects which page the user wants based on the URL)
Resulting HTML page

Remember how this was about MVC?

- So we have a layer which represents our **data model**.
- And we have a layer which **controls** how the data gets used and displayed.
- And we have a layer (here, just HTML, but we could abstract the HTML itself out into *templates*) which is simply the **view** that the user sees.
- **Model <-> Controller <-> View**
- That is, what we've designed is an application of the MVC (Model-View-Controller) **design pattern**


Model
(the data)
Stores data, allows modifications and retrieval.
MySQL Tables
Retrieved Data
Contr-oller
Manages the view; may modify data coming from view; updates the model.
View
Potentially abstracted through templates, but ultimately the web page itself.
Updates to Data
User Actions

What is a Design Pattern?

- In software development, one often runs into similar problems over and over
  - Though these problems can rarely be exactly categorized, often they can be generalized
- Experienced developers have developed general solutions for these general problems
  - These general solutions are called **design patterns**
- You may also be familiar with:
  - The Singleton pattern (if we have a class which we need only one instance of but want global access to)
  - The Factory pattern (if we want to wrap up the details and dependencies of creating an object)

Back to MVC

- MVC (again, Model-View-Controller) is a very general pattern. Unlike some design patterns, it does not exist to correct a deficiency in a programming language, but rather to describe a general solution to abstracting data manipulation by a user
- It was first used to design GUI applications:
    - If you have an app that has a perfectly workable API or console interface, treat that app as the **model**
    - To add a GUI, write an in-between **controller** layer (which updates the model and the view), and design the **view** (the actual GUI).

Why MVC?

- When building a dynamic website, enforcing a separation between data logic (i.e., SQL queries) and the view (the eventual HTML) is essential
  - Otherwise, you may end up with security holes, redundancies, etc
- Almost any way of separating the two will end up similar to MVC
  - Many frameworks and much research exist for MVC
  - RubyOnRails, Django, CakePHP all use MVC

Why bother?

- Many frameworks are based on MVC, so understanding it is crucial to working with them
    - Ruby on Rails, for example, bases its framework strongly around models, controllers, and views.
- If you develop your own framework, it doesn't have to be MVC
    - You should enforce database separation, however, and will probably end up with a template system -- both of which are present in MVC
- It's not the only option; however, it represents a solid design pattern that is commonly used on the web.

So, we have the general framework

- With the MVC pattern, we have an idea of how each part of the website fits together in the larger sense.
- However, there are still a lot of details:
  - HTML/CSS (we didn't spend much time on these)
  - JavaScript - useful for improved user interfaces
  - (My/Postgre/MS)SQL - how do we design a good database? how do we write efficient queries?
  - Security concerns - data validation, preventing XSS attacks, preventing SQL and header injection, etc.
  - And a lot more...

What next?

- We could take a break from tutorials to work on a project
  - Such as rebuilding the Webmonkeys website
- Could do tutorials on any of the topics on the previous slide, or on:
  - Ruby on Rails
  - Web design
  - HTML5