Models

lesson #basic03

James L. Parry
B.C. Institute of Technology

Agenda

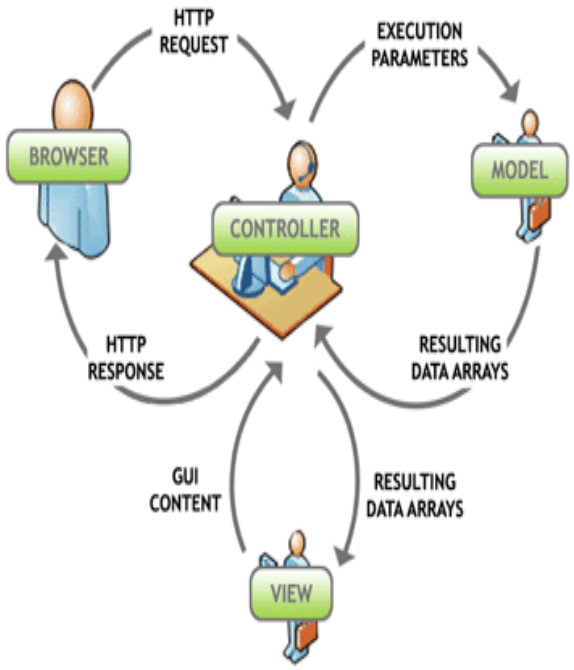
1. Model Patterns
2. Kinds of Models
3. CodeIgniter Models
4. Component Models
5. Object Relational Mapping
6. Database Utilities
7. Model Conventions
8. Database Configuration

MODEL PATTERNS

Models are the "model" component of the MVC design pattern.

The figure to the right shows this from the perspective of a webapp.

Fundamentally, models encapsulate data sources.

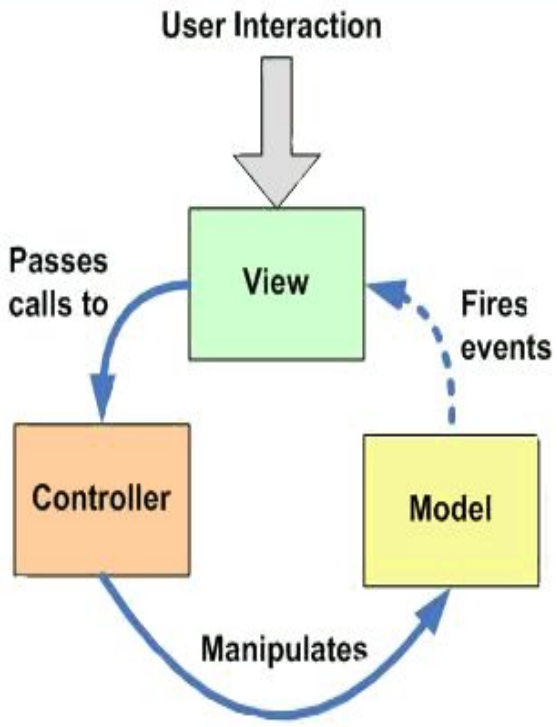


Classic Model-View-Controller

The figure to the right presents a "classic" view of the MVC pattern, from a programming perspective.

A user interacts with a **view**, using view components such as links, buttons, and form fields to initiate a request to a controller.

The "tricky" part? The view is supposed to be bound to a model as an event handler; when model state changes, an event is fired, and the view would respond to the event, updating its presentation as warranted.



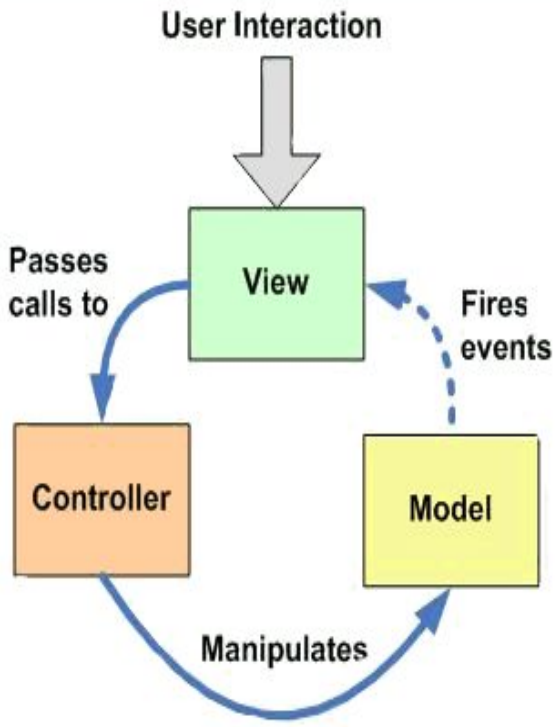
Model-View-Controller

Classic MVC and the Web

The "classic" MVC has two issues from a webapp's perspective:

- Accessing a model inside a view is legal, but considered a poor practice
- Propagating an event in a distributed system is awkward

The next two slides present solutions to these.



Model-View-Controller

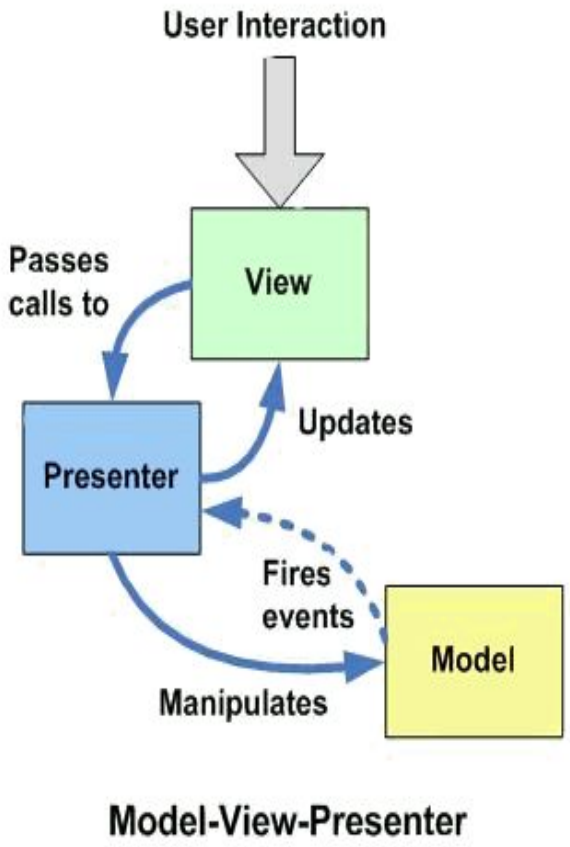
Model-View-Adapter

The model-view-adapter design pattern has the controller handle all interaction with a model, and pass data to a view through parameters.

The view is unaware of the source of the data it presents.

True event handling is still awkward with PHP, as our objects are not memory-resident.

This pattern is also called model-view-presenter.



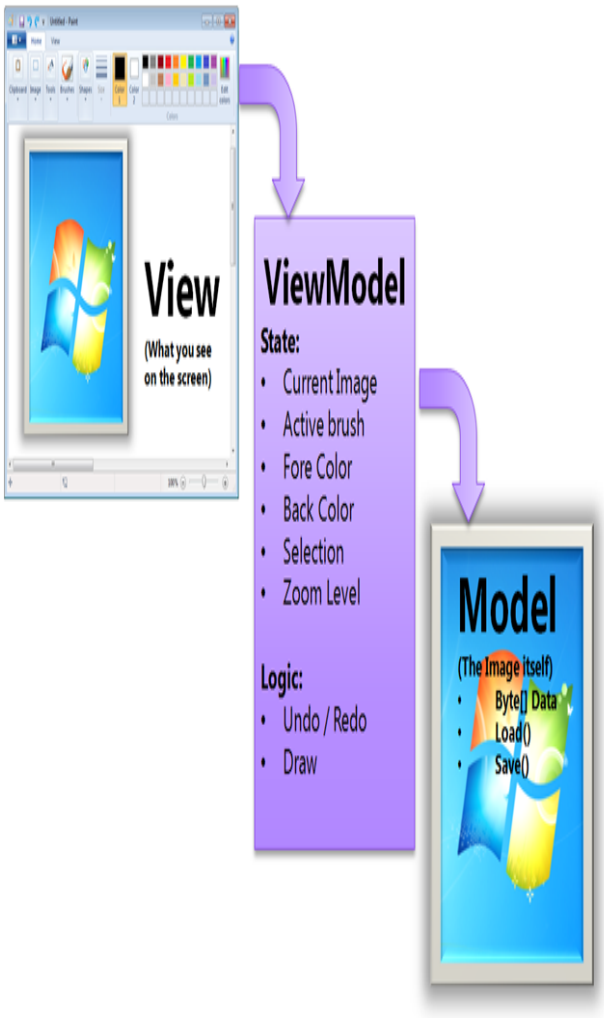
Model-View-Presenter

Model-View-ViewModel

The model-view-viewmodel design pattern has an explicit component to handle the model and related events, while keeping the view itself unaware of the source of its data.

In a webapp, this could be implemented by data-aware "widgets", typically programmed using Javascript.

This pattern is common in enterprise applications, and supported in ASP.net and JavaServerFaces.



KINDS OF MODELS

What is a model?

1) Models represent data sources <data access layer>

2) Models represent entities <domain model>

You've seen models corresponding to RDB tables as data sources. Let's dive deeper!

Domain Models

Domain models are awkward to enforce in PHP, without strong typing. They are normal in Java & C##.

For instance, Java supports JavaBeans (serializable, standard accessor and mutator methods) for generic domain models.

Java also supports Enterprise JavaBeans, following a managed, server-side component architecture. These can be session based, message-based, or persistent.

Active Record Models

The active record design pattern suggests that domain models have collection-centric methods in addition to entity property accessor/mutators.

This suggests that a model include

- `add($record)` – add a new record to the collection
- `get($key)` – retrieve a keyed record
- `update($record)` – replace a record
- `delete($key)` – delete a record

Some additional common model methods that support this:

- `create()` – return a suitable empty object
- `all()` – return all records
- `some(...)` – return some of the records
- `size()` – return the number of records

Active Record in Practice

Given how easy it is to inject or eliminate properties in PHP objects, many developers use entity properties directly, rather than attempting to follow the rigor of other language domain models

Given the way that PHP has evolved, many developers will also provide methods that return or work with associative arrays, using them to store entity properties.

This means doing something like `$customer->name` rather than something like `$customer->getName()`

Active Record in Practice /2

If adding collection centric methods to a "conventional" domain model, it is a good idea for your model to have collection-specific properties too, for instance the name of a database table and the fieldname used as a primary key.

Regardless of any accessors or CRUD methods you might have, make sure your model provides suitable methods for extracting whatever data your controllers need to pass on to a view!

This means having properties for the table name that a model is bound to, or for the XML document used to record state.

Data extraction methods,not related to a domain model, could be something like `findNewestBlogEntry()` or `processOrder($items)`

Note that these suggestions run counter to many "purist" perspectives ... PHP tends to be more relaxed!

Models Can Be Used For... ?

Models can be used to encapsulate many different sources of data, as you will see in the next few slides :)

Beyond relational database tables, we will take a quick look at encapsulating XML documents, file system folders, and even services.

These different kinds of models will be revisited later in the course, when we talk about XML and services.

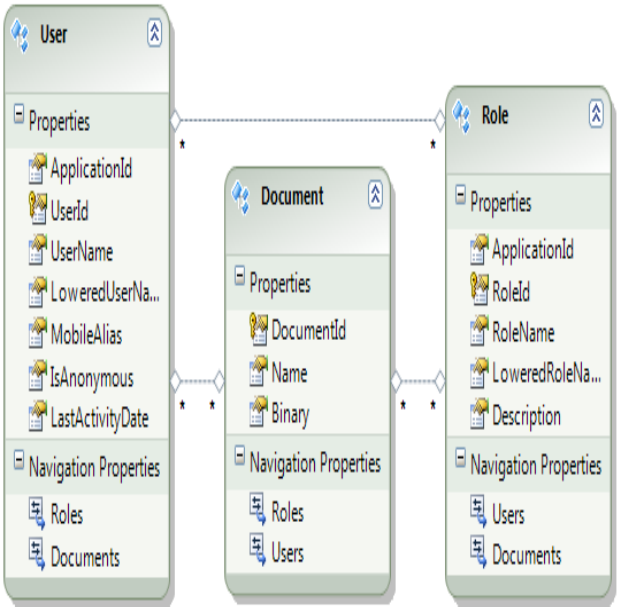
Models for RDB Tables

You would normally have one model per table in the relational database you are using.

CRUD methods in your model would deal with rows in the RDB table.

Reasonable model management properties in this kind of model would usually include the table name and the field name to use as the primary key.

Clearly 3 models...



Sample Model for an RDB Table

Referring to the contacts example from week 2, `application/core/MY_Model.php` is a sample base model you might use. `models/Contacts.php` is a model using the RDB base model. The following provides access to all the CRUD methods defined in MY_Model, making it easy to follow the active record pattern.

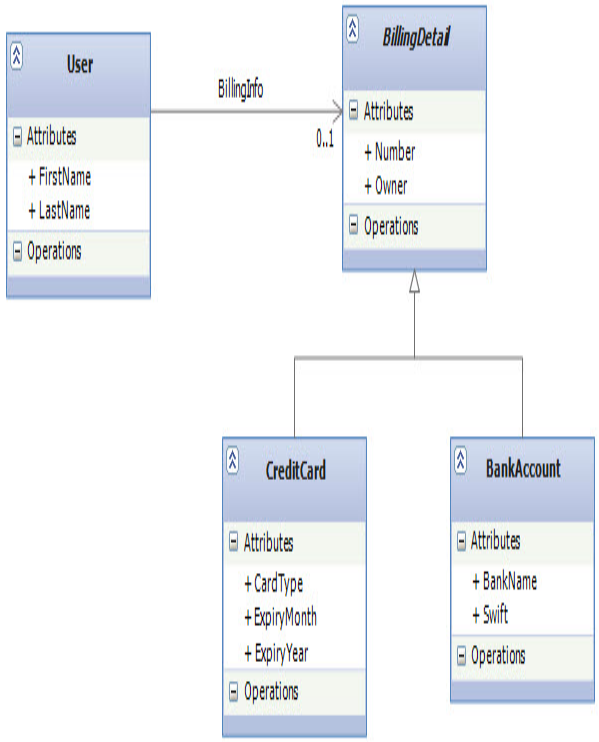
```
class Contacts extends MY_Model
{
  function __construct() {
    parent::__construct();
    $this->setTable('contacts',
'ID');
  }
}
```

Models for XML

You could use a model for a suitably structured XML document, for instance one where each child of the root element stored the state of an entity. You would need to choose a convention for the "primary key" in such a case, for instance the "id" attribute of an entity's element.

CRUD methods in your model would deal with children of the root element, and would traverse each child's attributes and elements to build a record object (or associative) array that looked like it came from an RDB table.

Clearly 3 models, or maybe 4, or just 2? ...



Models for XML /2

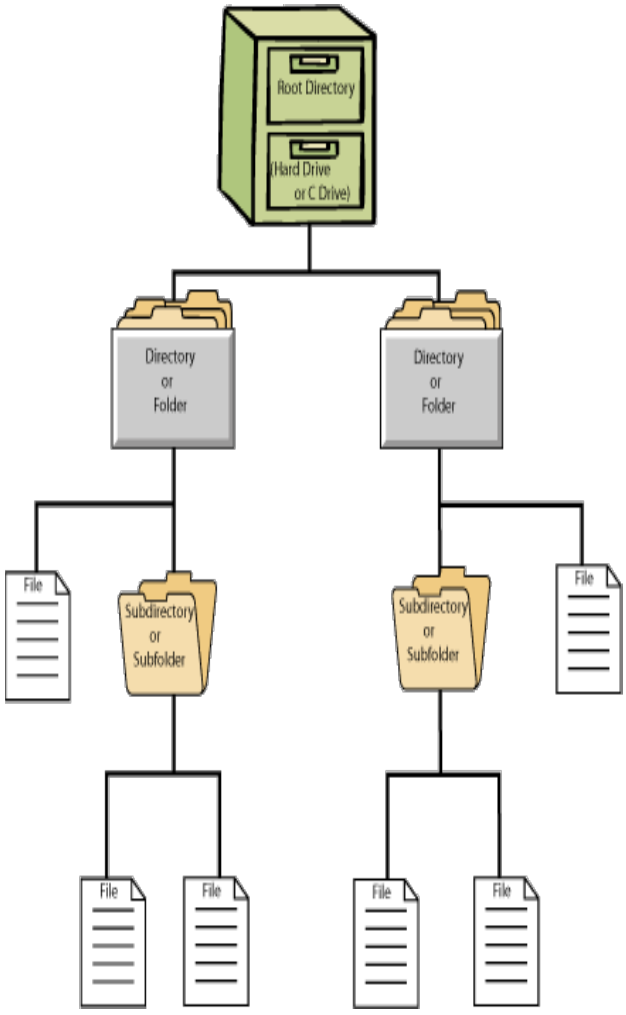
Management properties needed to build/rebuild the XML document for saving might include the document name, The element name of the root element, and the name of the attribute or nested child element to use as a "primary key".

Note that this approach could lend itself to using a DTD or schema for validation of "record" structure and content.

Models for Folders

You could use a model to encapsulate the contents of folders in your filesystem. The folder and subfolders would not have to be inside your webapp - they could be anywhere!

Such a model could be used for the files themselves, or to mimic RDB table rows through flat file representations of records.



Models for Folders /2

Management properties suited to this use might include the base folder name for your "collection", and perhaps the default extension/filetype. A filename itself might be considered a "primary key".

An example of this kind of model would be to provide access to a folder of images that support an image gallery. The `all()` method might return a list of image descriptor objects. You might use subfolder names as categories.

Models for Services

You could use a model to encapsulate a service, mapping the conventional CRUD methods to something more appropriate for a service. This isn't the only way to access a service, or even the best way - just one approach.

Your model would need some properties to handle the service configuration, for instance the endpoint of a remote service.

A sample method mapping is shown to the right. This is a foreshadowing of RESTful services :)

- `add($record)` sends a POST(x) to the service
- `create()` sends a POST() to the service
- `get($key)` sends a GET(x) to the service
- `all()` sends a GET() to the service
- `update($record)` sends a PUT(x) to the service
- `delete($key)` sends a DELETE(x to the service

CODEIGNITER MODELS

CodeIgniter leans more to the data access layer flavor of model, rather than the domain flavor of model.

Models in CodeIgniter are found in the `application/models` folder.

The model class name should have the first letter capitalized, and the filename should match that. You can get away with breaking this rule on Windows, but your webapp will break on Linux (the model's source file will not be found).

CI_Model

Each model should extend CI_Model, or your base model if you provide one.

Models need to be loaded before use. This can be done explicitly, using `$this->load->model('customers');` or it can be implied by configuration in `application/config/autoload.php`, with `$autoload['model'] = array('customers',...);`.

Once loaded, the model becomes a property of your controller, as in `$this->customers->...`.

Implementing Models

Some developers prefer to work closely with database drivers, writing their own SQL statements, as in

```
$results =
$this->db->query(...);
```

Other developers prefer to use the Query Builder for a more O-O approach to database manipulation, as in

```
$model->where(...);
$model->limit(...);
$results = $model->get();
```

Using a Base Model

Many developers use a base model (`application/core/MY_Model`) to provide a consistent implementation of the base model methods they want for their webapp. This typically includes all of the CRUD methods, regardless of their style (driver or query builder).

Your webapp models would extend the base model, and implement usecase-specific methods.

An example using a base model:

```
class Sales extends MY_Model {
  function __construct() {
    parent::__construct();
  }

  function new_order(...) {}
  function add_item(...) {}
  function calc_tax(...) {}
  function receipt(...) {}

}
```

Misusing a Base Model?

The CodeIgniter core classes can be extended by having a `MY_classname.php` inside `application/core`. You might have noticed, with `application/core /MY_Controller.php`, that the class name inside that source file does not *have* to match the MY_... naming convention. We saw that with our original example, with the `Welcome` controller extending `Application`.

CodeIgniter instantiates a singleton when a model or other core class is loaded, which males it awkward to deal with interfaces or abstract classes. However, we can exploit the general treatment of included source files by having more than one thing inside one.

Here is a Fancy Base Model!

Here is a "fancy" model. The source file includes an `Active_record` interface, as well as a `MY_Model` class. You could use this as a sort of template for writing adapters for other kinds of models (folders, XML, etc).

As long as one of your models, extending MY_Model, is loaded first, the `Active_record` interface will be accessible to you.

You could add additional models to the file, for instance `XML_Model` or `Folder_Model`, each implementing `Active_record`. Note that this is not necessarily a "best" practice", just a "possible" one, exploiting the current CodeIgniter. There will likely be better and more proper ways to do this with the next version!

Example Working With DB Driver

```
$query = $this->db->query('SELECT name, title, email FROM
customers');

foreach ($query->result() as $row) {
  echo $row->title;
  echo $row->name;
  echo $row->email;
}

echo 'Total Results: ' . $query->num_rows();
```

Example Working With Query Builder

```
$query = $this->db->get();

foreach ($query->results() as $row) {
  echo $row->title;
  echo $row->name;
  echo $row->email;
}

echo 'Total Results: ' . $query->count_all();
```

COMPONENT MODELS

Component models encapsulate entities with classes that follow well-known conventions.

The best known component model is the JavaBean, originally from Sun Microsystems.

PHP does not have a similar convention, but many enterprise developers will assume that you know it, and possibly that you will try to follow it where practical.

JavaBeans

JavaBean rules, in a nutshell:

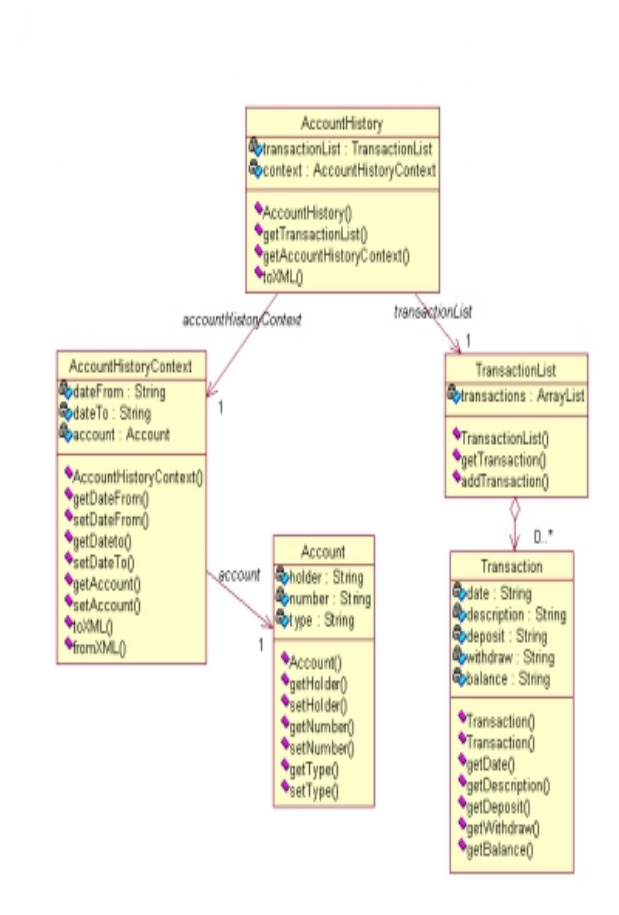
- No argument constructor
- Private fields
- Public accessor methods, a.k.a. getters
- Public mutator methods, a.k.a. setters

Common JavaBean practices:

- Convenience constructors
- Equality testing, `equals(object)`
- Text representation, `toString()`

Component Model Diagrams

Here is a typical UML class diagram showing some related JavaBeans.

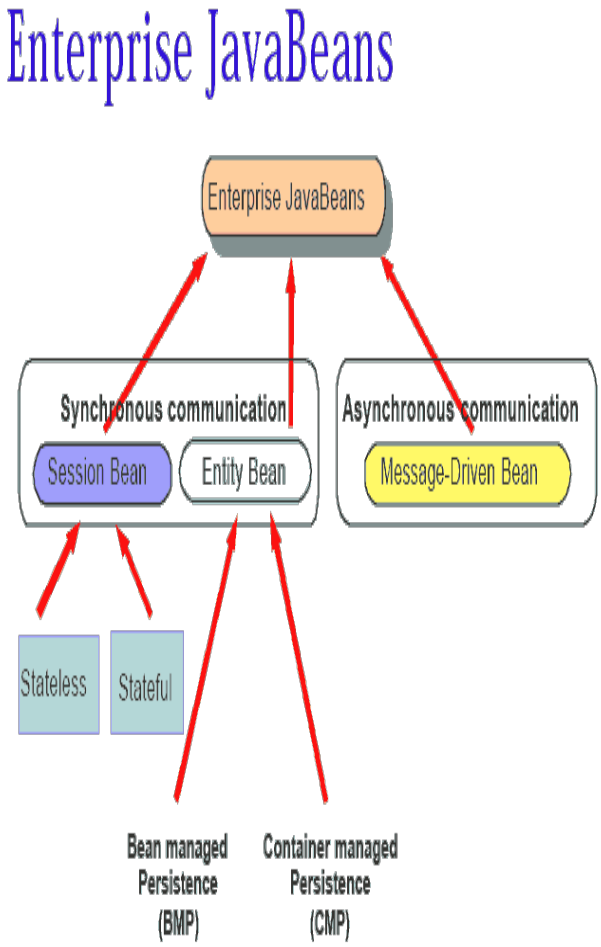


Enterprise Component Model

Building on JavaBeans, Enterprise JavaBeans provide a much more robust set of conventions, including the ability to handle persistence.

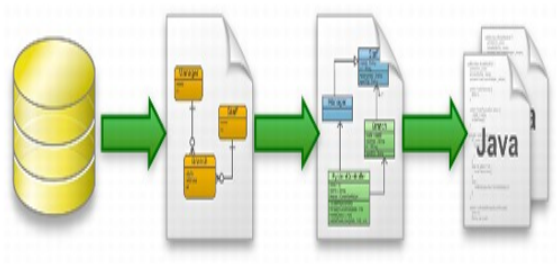
Without getting too carried away, the EJB inheritance hierarchy is shown to the right.

Why do you care? Many enterprise systems assume a similar infrastructure in webapps they work with or interact with.



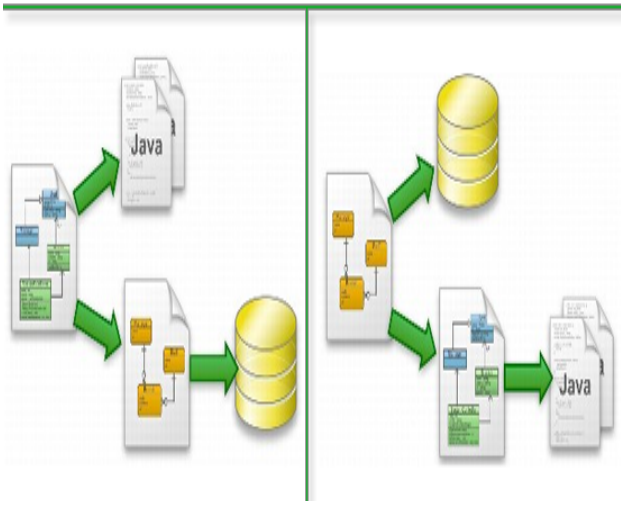
OBJECT-RELATIONAL MAPPING

Object-relational mapping is a programming technique or convention for converting data between an object model/state and a relational database model/state.



It is usually implemented by a tool or package which:

- From a database schema, generates an ERD, class diagram, and then code
- From a class diagram, generates code, an ERD and a schema
- From an ERD, generates a schema, and a class diagram and code



ORMs, Really...

In the PHP environment, object-relational mapping is typically effected by a third-party tool or plugin.

These tools attempt to automatically create the robustness of JavaBeans with the persistence of Enterprise JavaBeans.

These are notoriously awkward to implement properly.

Several such packages are available for CodeIgniter: Doctrine, Eloquent, and Propel. You're on your own using them, though - not part of this course!

Objects From Results

When PHP returns a result set from a relational database query, it usually does so as an iterable collection of row objects.

Each row object has properties corresponding to the table column names that the data came from.

This is *not* ORM, but it may suffice for many purposes.

If a row object is cast as an array, the result is an associative array with the table column names as indices.

Many PHP frameworks consider the objects and/or associative arrays returned by the database layer to be a simple form of ORM.

DATABASE UTILITIES

CodeIgniter has two classes specifically to manipulate databases.

The Database Forge class helps you work with metadata.

- Create or drop entire databases
- Create, drop or modify tables

The Database Utilities class helps you work with metadata.

- List databases
- Backup or export databases
- Repair or optimize databases
- Extract or convert records (CSV, XML)

MODEL CONVENTIONS

Use entity-related naming!

use the plural of an entity name for a table & model name. An example would be the `Posts` model for the `posts` table.

Use the singular of the entity name for a CRUD controller, for instance...

- `Post`
- `Post/view/x`
- `Post/add`

Avoid controllers and models with the same class name. Only one will be loaded. Namespaces (coming in CI4?) address this issue.

DATABASE CONFIGURATION

Your database access is configured by settings in `application/config/database.php`.

Read the Config class writeup carefully, as you want to use subfolders inside `application/config` to hold environment specific database configuration, for instance the username, password and database name to use when your webapp is deployed. It is a good practice to "git ignore" any such subfolders, so that you don't share confidential information on a public repository.

```
$db['default'] = array(
  'dsn' => '',
  'hostname' => 'localhost',
  'username' => 'root',
  'password' => '',
  'database' => 'database_name',
  'dbdriver' => 'mysqli',
  'dbprefix' => '',
  ...
  );
```

Database Conventions

For our purposes, I will expect your database configuraiton to work seemlessly when I pull your labs or assignments for marking. What this means:

- Make sure your default database configuration for "development" uses the username "root" with no password
- If I specify a database name, please use that, as I will have a test database already setup
- Use config subfolders, not commited to your repo, for any local or secure settings you need.

"Submission" Conventions

If you use a database in a project, clearly indicate so in your repo readme, and provide a SQL dump (gzipped) of your database, including dropping and tables you use before creating them, and including the insert statements to populate your database, as the file `setup.sql.gz` in either the root of your project or directly inside a `data` folder in your project. If you provide both, I will use the one in your project root.

If your webapp breaks because it is the wrong one, you're fired.
If you don't provide a database dump, and I cannot run your webapp at all, you're fired.

Congratulations!

You have completed lesson #basic03: Models

If you would take a minute to provide some feedback, we would appreciate it!

The next activity in sequence is: basic04 Views

You can use your browser's back button to return to the page you were on before starting this activity, or you can jump directly to the course homepage, organizer, or reference page.