Domain Driven Design
With ZF2 Intro

As of Zend Framework 2.*
Who Am I?

• Ralph Schindler (ralphschindler)
  ▪ Software Engineer on the Zend Framework team
    • At Zend for 5 years
    • Before that TippingPoint/3Com
  ▪ Programming PHP for 14+ years
  ▪ Live in New Orleans, LA.
    • Lived in Austin, Tx for 5 years
This Webinar

- Brief words on Modeling
- General introduction To Domain Driven Design concepts
- ZF2 Tools
- Look at a real application:
  - https://github.com/ralphschindler/DominDrivenDesignZF2Intro
What Is Modeling?

• (Loosely defined) M in the MVC

• From Wikipedia:
  ‣ A controller can send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document). It can send commands to the model to update the model's state (e.g., editing a document).

  ‣ A model notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands. A passive implementation of MVC omits these notifications, because the application does not require them or the software platform does not support them.

  ‣ A view requests from the model the information that it needs to generate an output representation.
What does that mean really?

- In PHP, you can generally think of it like this:
  - Controllers interact with environment
    - $_POST, $_SERVER, $_GET, environment variables, etc
  - Views are responsible for display concerns
    - What does my HTML look like
    - As I iterate this object or array, how do I format it
    - How do I escape data for consumption in a web browser
  - Which leaves the Model...
The Model is ...

• A set of characteristics:
  ‣ The core of your business problem
  ‣ Data & the persistence of that data
  ‣ UI agnostic (HTML and JSON agnostic)
    • aka: View agnostic concerns / Not a view
    • Models don’t have an opinion on how they are displayed
  ‣ Environment agnostic (CLI vs. Browser)
    • aka: Controller agnostic concerns / Not a controller
    • Models don’t have an opinion on how they are consumed
    • ...
... continued,

In OO terms:

- OOM: Object oriented modeling
- A way of conceptualizing a problem domain into classes and objects to better manage their complexity, to simplify it
- Present business object workflows in easy to understand and consume API
Domain Driven Design

- Introduced by Eric Evans
  - Some collaboration with Martin Fowler
- Domain Driven Design (2004)
- Also: Domain Driven Design Quickly
  - a good summary in pdf (google it)
Domain Driven Design (Wikipedia)

• Core Definitions

  ▶ **Domain**: A sphere of knowledge (*ontology*), influence, or activity. The subject area to which the user applies a program is the domain of the software.

  ▶ **Model**: A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

  ▶ **Ubiquitous Language**: A language structured around the domain model and used by all team members to connect all the activities of the team with the software.

  ▶ **Context**: The setting in which a word or statement appears that determines its meaning.
Domain Driven Design (Wikipedia)

Success For This Approach In A Project

- The domain is not trivial

- The project team has experience and interest in Object Oriented Programming/Design

- The project has access to domain experts

- There is an iterative process in place
Building Blocks / Patterns

• Entity vs. Value Object vs. Value
  ▸ An *Entity* has an identity and a *value object* does not.
  ▸ Both are generally POPO’s (Plain old PHP objects).
  ▸ By definition, *value objects* are identity free and immutable.
  ▸ *Values* are simply put, any scalar in PHP (for all intents and purposes).
  ▸ Two separate *Entities* can share the same reference to a *Value Object*. 
Entity Scenarios (wikipedia):

- Example: Most airlines distinguish each seat uniquely on every flight. Each seat is an entity in this context. However, Southwest Airlines (or EasyJet/RyanAir for Europeans) does not distinguish between every seat; all seats are the same. In this context, a seat is actually a value object.
// Most Airlines
class Seat {
    protected $number;
    protected $isExitRow = false;
    public function __construct($number, $isExitRow = false) {
        $this->number = $number;
        $this->isExitRow = $isExitRow;
    }
    public function getNumber() {
        return $this->number;
    }
    public function isExitRow() {
        return $this->isExitRow;
    }
}

// Boeing 737
foreach (range(1, 35) as $n) {
    foreach (range('A', 'F') as $l) {
        $flight->addSeat(new Seat($l . $n));
    }
}
Building Blocks / Patterns

// Southwest
class Seat {
}

// Boeing 737
foreach (range(1, 175) as $n) {
    $flight->addSeat(new Seat());
}
Value Object

- PHP's DateTime object does not qualify:

```php
class DateTime {
    public function modify(/* string */ $modify);
    public function something();
    public function add(DateInterval $interval);
}
```

- Your own will:

```php
class Date {
    public function getYear();
    public function getMonth();
    public function getDay();
    private function __get($name);
}
```
Building Blocks / Patterns

• Aggregate Root Example

  Example: When you drive a car, you do not have to worry about moving the wheels forward, making the engine combust with spark and fuel, etc.; you are simply driving the car. In this context, the car is an aggregate of several other objects and serves as the aggregate root to all of the other systems.
Building Blocks / Patterns

• Aggregate & Aggregate Root
  ‣ A *collection of objects* that are bound together by a root entity, otherwise known as an *aggregate root*. The aggregate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members.

  ‣ In PHP speak:
    • object graph
    • objects with references to others
Building Blocks / Patterns

```php
class PlaylistService {
    protected $playlistRepo;

    public function __construct(PlaylistRepositoryInterface $playlistRepo) {
        $this->playlistRepo = $playlistRepo;
    }

    public function findAll() {
        return $this->playlistRepo->findAll();
    }

    public function findById($id) {
        return $this->playlistRepo->findById($id);
    }
}
```

Playlist Name: `<?php $playlist->getName(); ?>`
Tracks:

```php
<?php foreach ($playlist->getTracks() as $i => $track): ?>
    <?php $i++; ?>: <w> <?php $track->getTitle(); ?>
by <b><?php $track->getArtist()->getName(); ?></b>
on <i><?php $track->getAlbum()->getTitle(); ?></i>
(<?php $track->getAlbum()->getReleaseDate(); ?>)
<?php endforeach; ?>
```
Building Blocks / Patterns

• Repository
  ‣ methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged

  ‣ “implementations may be easily interchanged”
    • of secondary importance in PHP
Sidenote: Interfaces

• When to use interfaces:
  ‣ When an alternate implementation is conceivable and likely
interface TrackRepositoryInterface {
    // @return Track[]
    public function findAll();
    public function findById($id);

    public function store(Track $track);
    public function remove(Track $track);
}

// IMPLEMENTATION
class DbTrackRepository implements TrackRepositoryInterface {
    public function __construct(TrackDbMapper $mapper) {}
    /** ... **/}

// USAGE
$trackRepo = new DbTrackRepository($services->get('TrackMapper'));
$tracks = $trackRepo->findAll();
foreach ($tracks as $track) {
    // do something interesting
}
Building Blocks / Patterns

• Factory

  ‣ methods for creating domain objects should delegate to a specialized Factory object such that alternative implementations may be easily interchanged.

  ‣ using new Entity();
  ‣ using EntityFactory::create()
  ‣ using (new EntityFactory())->createEntity()
Building Blocks / Patterns

- Service
  - First, a word on services ...
Other: Services

• Services (in multiple contexts)
  ▸ Service Layer: separate abstraction layer between controllers and models
  ▸ Model Services: (DDD) A place where "workflows/functions that have no natural place in a value object/entity"
  ▸ Dependency Injection / Application Architecture: shared objects, dependencies (Service Locator)
Building Blocks / Patterns

• Services in DDD
  
  Service: When an operation does not conceptually belong to any object. Following the natural contours of the problem, you can implement these operations in services.
// This is a "Service Layer" class
// An abstraction layer between controllers and your models
// You might do this so that your services are portable between different kinds of applications (think an API/web-service centric application)

class PlaylistService
{
    protected $playlistRepo;

    public function __construct(PlaylistRepositoryInterface $playlistRepo)
    {
        $this->playlistRepo = $playlistRepo;
    }

    public function findAll()
    {
        return $this->playlistRepo->findAll();
    }

    public function findById($id)
    {
        return $this->playlistRepo->findId($id);
    }
}
Patterns: The tools in our toolbox

• Different patterns describe a particular abstraction, that might suit our need

• Covered in a previous Webinar:
  ▸ TableGateway, RowGateway
    ▹ Implemented by Zend\Db
  ▸ ActiveRecord
  ▸ Mapper
  ▸ Lazy Loading & Lazy Loading Via PHP Closure

• Domain Driven Design patterns:
  ▸ Repository
  ▸ Entity, Value Object, Value
  ▸ Other briefly for context
TableGateway & RowGateway

- Implemented in Zend\Db

- TableGateway, specifically, can be used:
  - Directly as the gateway to “model data”
    - No abstraction: “Models” are really associative arrays in this scenario
  - Directly as the data access for a Repository
    - 1 level of abstraction: Essentially as a mapper
  - Or as the implementation detail of a Mapper
    - 2 levels of abstraction: Repository > Mapper > TableGateway
class ArtistMapper
{
    public function mapArrayToArtist(array $data, Artist $artist = null)
    {
        $artist = ($artist) ?: new Artist;
        $artist->firstName = $data['first_name'];
        $artist->lastName = $data['last_name'];

        $album = new Album;
        $album->title = $data['album_1_title'];
        $artist->albums[] = $album;
        return $artist;
    }
}
class DataMapper {
    protected function mapPlaylistRowToObject(array $row) {
        $playlist = new Playlist;
        $playlist->setId($row['id']);
        $playlist->setName($row['name']);
        $playlist->setTracks($this->lazyLoadTracksClosure($row['id']));
        return $playlist;
    }

    protected function lazyLoadTracksClosure($playlistId) {
        $dataMapper = $this; // php 5.3 hack, must be renamed
        return function () use ($playlistId, $dataMapper) {
            // ...
            return $tracks;
        };
    }
}
Lazy Loading Via Closure/Anon Func.

class Playlist {
    public function setTracks($tracks) {
        $this->tracks = $tracks;
    }
    public function getTracks() {
        if ($this->tracks instanceof \Closure) {
            $this->tracks = call_user_func($this->tracks);
        }
        return $this->tracks;
    }
}
Zend\Db In Modeling

• What does one need to know?
  ‣ Previous webinar:
    • Have an overall idea of the architecture:
      - Zend\Db\Adapter’s Drivers & Platform objects for Driver Abstraction
      - Zend\Db\Sql for Sql as OO as well as SQL abstraction
  ‣ Strengths of Zend\Db
    • The base TableGateway is a solid approach to an object per table
    • The Zend\Db\Sql\Select API is expansive and offers full a framework for full SQL abstraction
    • Zend\Db is not a modeling framework on its own
Hydration

• Part of Zend\Stdlib
• Moves data from an array to an object and visa-versa

```php
$playlistTable = new TableGateway(
    'playlist',
    $this->dbAdapter,
    null,
    // Zend\Stdlib\Hydrator\ClassMethods will do
    // $obj->setFirstName($arr['first_name'])
    // for each property on iteration
    new HydratingResultSet(new ClassMethodHydrator(), $playlistPrototype)
);```
Let’s build an app!

- **Code Location:**
  - https://github.com/ralphschindler/DomainDrivenDesignZF2Intro/

- **Problem Domain:**
  - There is money in sharing playlists online.
  - I am not sure what the business will be, but I know it centers around a playlist.
  - We need to be able to model Track, Artist and Album information.
  - We might want to be able to pull information from web services.
Thanks!

http://twitter.com/ralphschindler
http://framework.zend.com/zf2
http://github.com/zendframework/
http://github.com/ralphschindler