London Laravel

Interfaces and Repositories

13 Nov 2014

Example controller

```
class ContactsController extends BaseController
{
  public function all()
  {
    $contacts = \Contact::all();

    return View::make('contacts.index', compact('contacts'));
  }
}
```

This is a bad idea ™

- Tightly coupled code
- Tied to eloquent
- Not (easily) testable
- Not DRY
- Maintenance
- Bugs
- arggggggg!!

Repositories

- Acts as a service layer between your application and your DB layer
- This means you don't interact directly with your models
- Loosely coupled code
- Easily changeable DB layer

Repository example

```
Class ContactRepository
{
    public function all()
    {
        return \Contact::all();
    }

    public function find($id)
    {
        return \Contact::find($id);
    }
}
```

This is a better idea ™

```
class ContactsController extends BaseController
{
  public function all()
  {
    $contactsRepo = new ContactRepository()
    $contacts = $contactsRepo->all();

    return View::make('contacts.index', compact('contacts'));
  }
}
```

But there are still some problems

- Tightly coupled to a specific repo
- Avoid using the “new” word as much as possible

Dependency Injection

- Dependency injection is a software design pattern that implements inversion of control and allows a program design to follow the dependency inversion principle.
- An injection is the passing of a dependency (a service) to a dependent object (a client).
- What the f*&k does that mean?

Dependency Injection

- Instead of your classes directly creating an object (`new ContactRepository`), we inject that object into the constructor
- Class methods never (ever!?) use the “new” keyword
- Loosely coupled code (again...)
- Infinitely more maintainable

Interfaces

- Code to an Interface, not to an Implementation
- Managing dependencies in a large application
- Represents “what” a class can do, but not “how”
- Any class which implements an interface must provide an implementation of all methods
  - Enforces a contract
  - Results in loosely coupled code because you are not reliant on on the implementation

Interface example

```
interface ContactInterface
{
  public function all();
  public function store($data);
}

class ContactRepository implements ContactInterface
{
  public function all()
  {
    // some awesome code to retrieve all records
  }

  public function store($data)
  {
    // some awesome code to store a new contact
  }
}
```

DI example

```
class ContactsController extends BaseController
{
  protected $contacts;

  public function __construct(ContactInterface $contacts)
  {
    $this->contacts = $contacts;
  }
}
```

But how?

Laravel is awesome

```
App::bind('ContactInterface','ContactRepository');
```

A better better idea ™

```
class ContactsController extends BaseController
{
  protected $contacts;

  public function __construct(ContactInterface $contacts)
  {
    $this->contacts = $contacts;
  }

  public function all()
  {
    $contacts = $this->contacts->all();

    return View::make('contacts.index', compact('contacts'));
  }
}
```

Contact Interface

```
interface ContactInterface
{
    public function all();
    public function paginate($count);
    public function find($id);
    public function store($data);
    public function update($id, $data);
    public function delete($id);
    public function findBy($field, $value);
}
```

User Interface

```
interface UserInterface
{
    public function all();
    public function paginate($count);
    public function find($id);
    public function store($data);
    public function update($id, $data);
    public function delete($id);
    public function findBy($field, $value);
}
```

Job Interface

```
interface JobInterface
{
    public function all();
    public function paginate($count);
    public function find($id);
    public function store($data);
    public function update($id, $data);
    public function delete($id);
    public function findBy($field, $value);
}
```

Foo interface

Why are we repeating ourselves?

A better better better idea ™

```
interface BaseInterface
{
    public function all();
    public function paginate($count);
    public function find($id);
    public function store($data);
    public function update($id, $data);
    public function delete($id);
    public function findBy($field, $value);
}
```

Extend the BaseInterface

```
interface ContactInterface extends BaseInterface {}

interface UserInterface extends BaseInterface {}

interface JobInterface extends BaseInterface {}

Interface FooInterface extends BaseInterface {}
```

The best (?) idea ™

```
class BaseRepository
{
  protected $modelName;

  public function all()
  {
    $instance = $this->getNewInstance();
    return $instance->all();
  }

  public function find($id)
  {
    $instance = $this->getNewInstance();
    return $instance->find($id);
  }

  protected function getNewInstance()
  {
    $model = $this->modelName;
    return new $model;
  }
}
```

So clean....

```
class ContactRepository extends BaseRepository
{
    protected $modelName = 'Contact';
}

class UserRepository extends BaseRepository
{
    protected $modelName = 'User';
}
```

And thats it...

How about some eager loading?

```
class BaseRepository
{
  public function find($id, $relations = [])
  {
    $instance = $this->getNewInstance();
    return $instance->with($relations)->find($id);
  }
}

class ContactsController extends BaseController
{
  public function find($id)
  {
    $contact = $this->contacts->find($id, ['orders']);

    return View::make('contacts.show', compact('contact'));
  }
}
```

Multi Tenant

- Multi-tenant database introduces some complexity
- Example: we have multiple orders per user, and allow the user to view their own orders
- Ensure that they can't view someone else's orders

```
\Order::where('user_id', $userId)->all();
\Order::where('id', $orderId)->where('user_id', $userId)->find();
```

Noooooooooooooooooooooo!

Tenant Repository example

```
class BaseTenantRepository extends BaseRepository
{
    protected $tenantKey = 'user_id';
    protected $tenantId;

    protected function getNewInstance()
    {
        $model = $this->modelName;
        return $model::where($this->tenantKey, $this->tenantId);
 }
}
```

Tenant Controller

```
class OrdersController extends BaseController
{
    protected $orders;

    public function __construct(OrderInterface $orders)
    {
        $this->orders = $orders;
    }

    public function all()
    {
        $orders = $this->orders->all();

        return View::make('orders.index', compact('orders'));
    }

    public function show($id)
    {
        $order = $this->orders->find($id);

        return View::make('orders.show', compact('order'));
    }

}
```

Bonus Points

```
BaseAdminResoureceController extends AdminController
{
  protected $repositoryInterface;
  protected $viewPath;

  public function __construct($interface)
  {
    $this->repositoryInterface = $interface;
  }

  public function index()
  {
    $items = $this->repositoryInterface->all();

    return View::make($this->viewPath . '.index',
       compact('items'));
  }
}
```

Bonus Points

```
AdminOrdersController extends BaseAdminResoureceController
{
  protected $viewPath = 'admin.orders';

  public function __construct(AdminOrderInterface $interface)
  {
    parent::__construct($interface);
  }
}
```

Thanks!

Questions?