Laravel + AngularJS - Contact Manager

To complete this workshop you will need to have the following installed and working on your machine.

- Composer or Laravel Installer
- Larvel's Homestead

The following software is not required to complete this workshop; however, they are the tools that I will be using.

- iTerm or Terminal
- Sublime Text 2
- Sequal Pro
- Google Chrome
- Postman - REST Client (A Google Chrome Plugin)

Background

We are building a simple contact manager.

This contact manager will store a person's first name, last name, email, phone number and home address. We will be using Laravel to create a RESTful API, and using Angular JS on the client to consume that API, and display the results.

Starting

To start our project let's create a directory called `contactmgr` on our desktops:.

```
> cd ~/Desktop
> laravel new contactmgr
Crafting application...
```

Once the application has been created we will need to update our Homestead configurations.

```
> cd ~/Homestead
> vim Homestead.yaml
```

I am using VIM; however, if you are not comforatable with using VIM please choose an editor of your choice such as pico

```
> pico Homestead.yaml
```

We will be adding the following lines to the folders and sites sections:

```
folders:
    - map: PATH/TO/Desktop/contactmgr
    - to: /home/vagrant/contactmgr

sites:
    - map: contactmgr.app
    - to: /home/vagrant/contactmgr/public
```

Lastly, we will to modify our hosts file to point the domain contactmgr.app to our own machine.

```
> sudo vim /etc/hosts
```

On Windows, its located at

```
C:\Windows\System32\drivers\etc\hosts
```

We will add the following to the end of this file:

```
127.0.0.1 contactmgr.app
```

Now lets bring up your vagrant machine with

```
> cd ~/Homestead
> vagrant up --provision
```

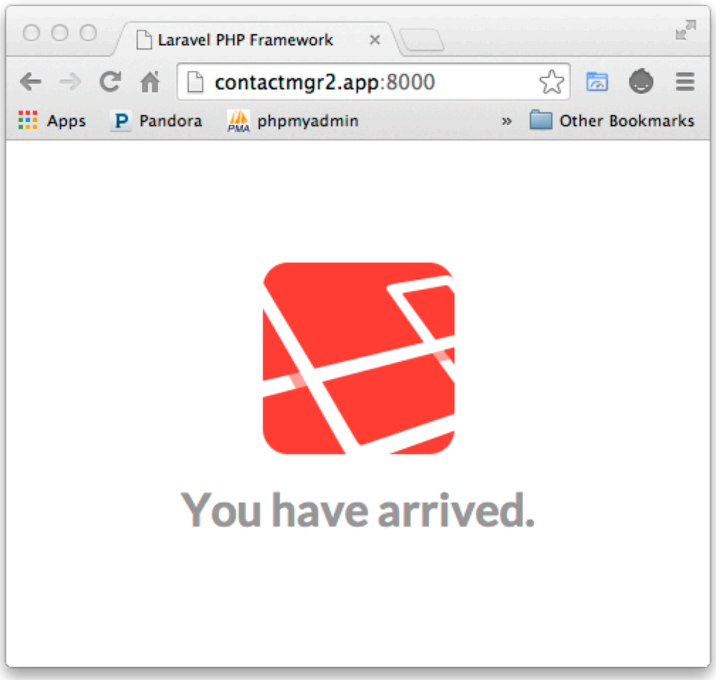
I have added the provision flag to force vagrant to reload the configuration files. Let's check that our contact manager files are showing up in vagrant.

```
> vm
# ssh vagrant@127.0.0.1 -p 2222
```

Now that we can see that our files are in vagrant, open Chrome and browse to

```
http://contactmgr.app:8000
```

You should now see



You have arrived.

Building Out Laravel

At this point we should have the Laravel framework installed, and the vargrant server running. With those tools in place we can start to build our the different parts of our application. We will start with Laravel.

Composer

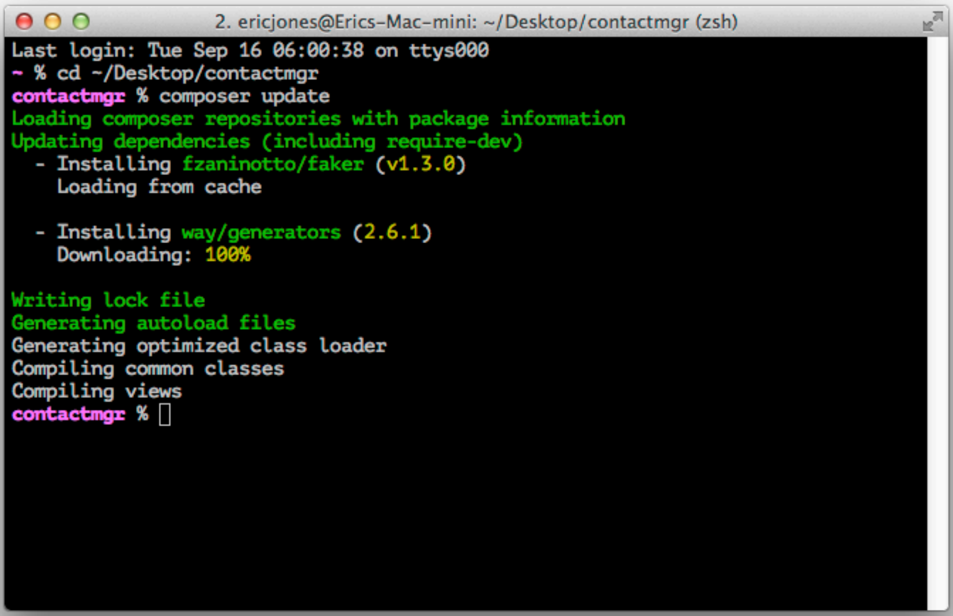
To help us with development I would like to use two packages, Jeffery Way's Generators Package, and PHP Faker. To include these packages to our project we need to add them to composer like so:

```
...
"require-dev": {
    "way/generators": "2.*",
    "fzaninotto/faker": "v1.3.0"
},
...
```

This will pull these two packages into our vendor folder, and allow us to use them in our project.

```
> composer update
```

You will now see something similar to this:



```
Last login: Tue Sep 16 06:00:38 on ttys000
~ % cd ~/Desktop/contactmgr
contactmgr % composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing fzaninotto/faker (v1.3.0)
    Loading from cache

  - Installing way/generators (2.6.1)
    Downloading: 100%

Writing lock file
Generating autoload files
Generating optimized class loader
Compiling common classes
Compiling views
contactmgr %
```

In order to use the generators package, we will need to tell Laravel about it.

Open `app/config/app.php`, and add the following to the providers array

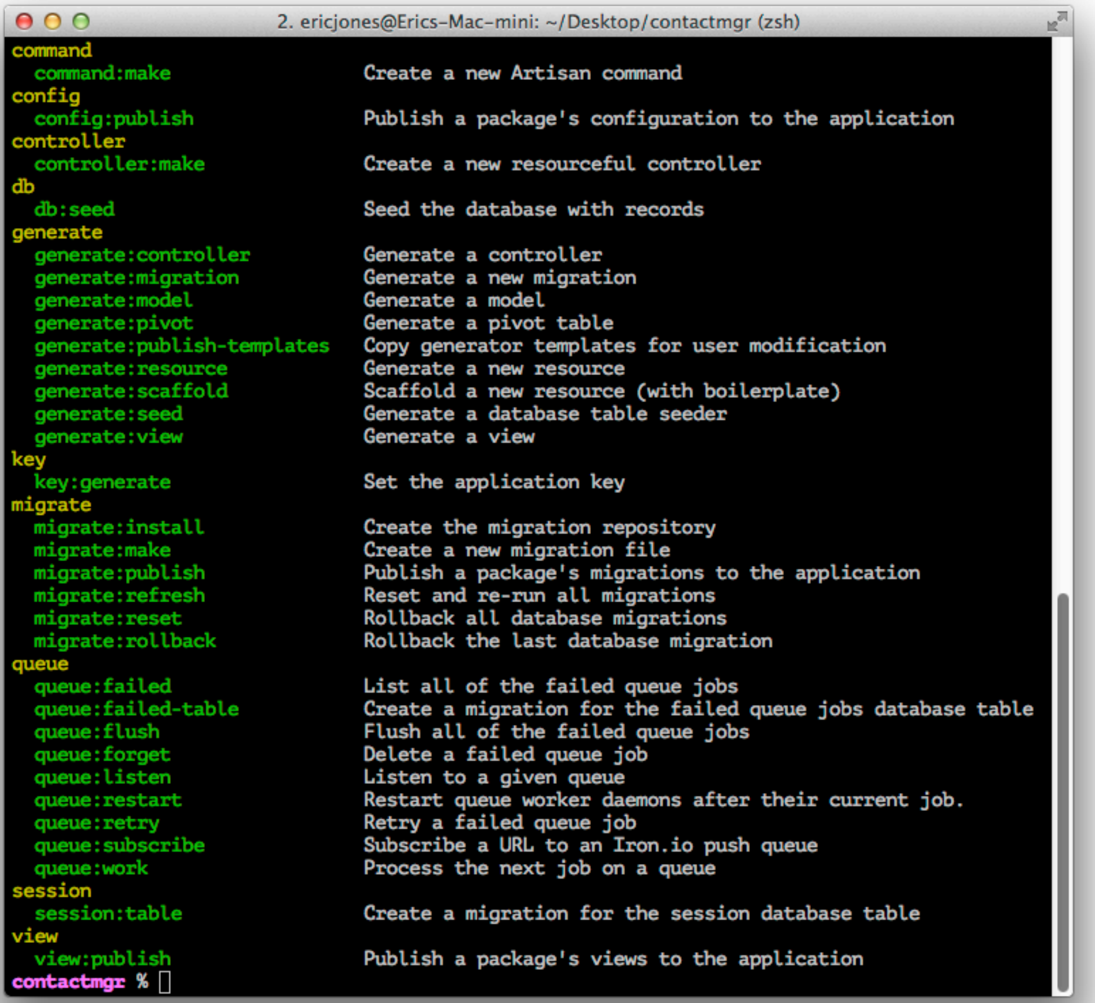
```
...
'providers' => array(
    ...
    # Third-Party Providers
    'Way\Generators\GeneratorsServiceProvider'
),
...
```

To check that this Provider is available to our command-line, let's do the following:

```
# If you are not in your project directory...

> cd ~/Desktop/contactmgr
> php artisan
```

And you should see the new commands



```
command
  command:make                Create a new Artisan command
config
  config:publish              Publish a package's configuration to the application
controller
  controller:make             Create a new resourceful controller
db
  db:seed                     Seed the database with records
generate
  generate:controller         Generate a controller
  generate:migration          Generate a new migration
  generate:model              Generate a model
  generate:pivot              Generate a pivot table
  generate:publish-templates  Copy generator templates for user modification
  generate:resource           Generate a new resource
  generate:scaffold           Scaffold a new resource (with boilerplate)
  generate:seed               Generate a database table seeder
  generate:view               Generate a view
key
  key:generate                Set the application key
migrate
  migrate:install             Create the migration repository
  migrate:make                Create a new migration file
  migrate:publish             Publish a package's migrations to the application
  migrate:refresh             Reset and re-run all migrations
  migrate:reset               Rollback all database migrations
  migrate:rollback            Rollback the last database migration
queue
  queue:failed                List all of the failed queue jobs
  queue:failed-table          Create a migration for the failed queue jobs database table
  queue:flush                 Flush all of the failed queue jobs
  queue:forget                Delete a failed queue job
  queue:listen                Listen to a given queue
  queue:restart               Restart queue worker daemons after their current job.
  queue:retry                 Retry a failed queue job
  queue:subscribe             Subscribe a URL to an Iron.io push queue
  queue:work                  Process the next job on a queue
session
  session:table               Create a migration for the session database table
view
  view:publish                Publish a package's views to the application
contactmgr %
```

While we are here let's add a "PSR-4" autoload declaration for our project. Go inside of the "autoload" declaration and add the following:

```
...
"autoload": {
    ...
    "psr-4": {
        "IndyDev\\": "app/IndyDev/"
    }
},
...
```

After which we will need to add the IndyDev folder inside of our app folder.

```
> mkdir ./contactmgr/app/IndyDev
> composer dump-autoload -o
```

The second command will create an optimized autoload file.

We are now done with composer.

Setting Up Your Database

Watch Out:

Laravel comes with the ability to configure your database based upon environments. So let's determine which environment we are in.

From Local Machine:

```
> php artisan env
Current application environment: production
```

From Vagrant Machine:

```
> php artisan env
Current application environment: local
```

At first this was a little confusing to me, but if we are running our application through Vagrant our configurations should be done within the local config folders.

Open up the file `app/config/local/database.php`. You should see two databas connection options: mysql, and postgresql. We will be using the mysql connection.

Because Homestead has already set up our database user account all we need to do is change the database name.

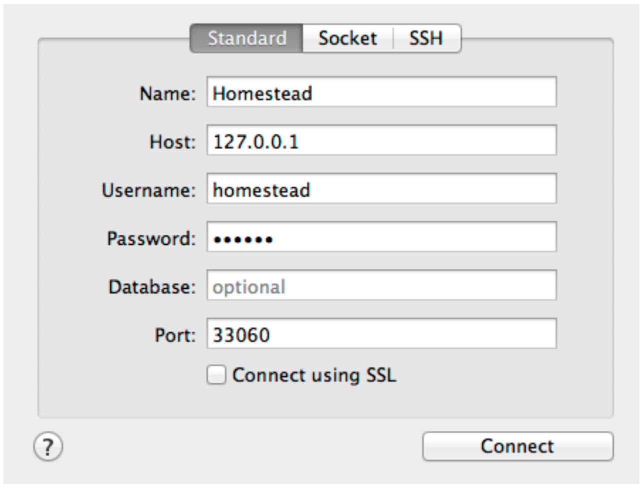
```
'mysql' => array(
    ...
    'database'  => 'contactmgr',
    ...
),
```

Connecting To Your Database

I will be using **Sequel Pro** to connect to my database; however, feel free to use whatever application you feel comfortable with using.

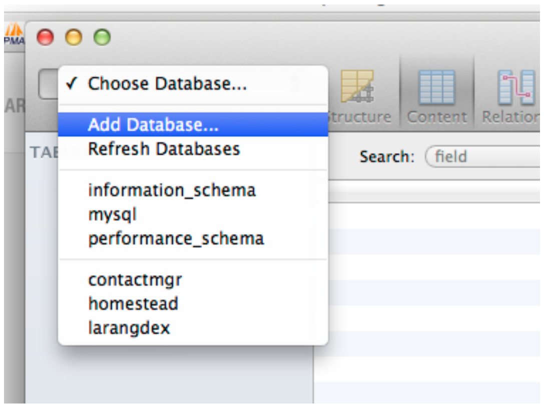
To connect to your Vagrant Database from your local machine, you will need to use the following credentials.

```
Host: 127.0.0.1
Username: homestead
Password: secret
Port: 33060
```



Name: Homestead
Host: 127.0.0.1
Username: homestead
Password: ••••••
Database: optional
Port: 33060
Connect using SSL

Now that you are connected to your database, let's create the contactmgr database.



Finish creating your database by giving it the name, 'contactmgr'.

With our database created, we can test the connection between our application and database by running the following command from within our Vagrant Machine:

```
> php artisan migrate:install
Migration table created successfully
```

This will create our migrations table, and you can confirm that by reviewing the database we previously connected to.

Create The Contacts Table

When we included the Generators package into our project, we gave ourselves an easier way to create migrations.

```
> php artisan generate:migration create_contacts_table --fields="firstName:string, lastName:string:nullable,
birthday:date:nullable, email:string:nullable, phone:string:nullable, street1:string:nullable, street2:string:nullable,
city:string:nullable, state:string:nullable, zip:string:nullable"
```

Let's break down this command into three parts:

```
1: php artisan generate:migration
```

This calls the migration command from the generators package that we included into our project from composer.

```
2: create_contacts_table
```

The generators package includes logic that understands from this string we want to literally create a contacts table.

```
3: --fields="FIELD:TYPE:OPTION, FIELD:TYPE:OPTION"
```

The generators package allows us to define the field name, field type and any field options.

To create our firstName field we would create this:

```
--fields="firstName:string"
```

This would create a firstName field that is a varchar and not null.

To give a better view of the fields I have them listed on each line.

```
firstName   : string,

lastName    : string    : nullable,

birthday    : date      : nullable,

email       : string    : nullable,

phone       : string    : nullable,

street1     : string    : nullable,

street2     : string    : nullable,

city        : string    : nullable,

state       : string    : nullable,

zip         : string    : nullable
```

Once run the generator will create a migration that looks similar to this in folder `app/database/migrations` : (Note: I added the line breaks for easier readability)

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateContactsTable extends Migration {

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('contacts', function(Blueprint $table)
        {
            $table->increments('id');

            $table->string('firstName');

            $table->string('lastName')->nullable();

            $table->date('birthday')->nullable();

            $table->string('email')->nullable();

            $table->string('phone')->nullable();

            $table->string('street1')->nullable();

            $table->string('street2')->nullable();

            $table->string('city')->nullable();

            $table->string('state')->nullable();

            $table->string('zip')->nullable();

            $table->timestamps();
        });
    }


    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('contacts');
    }

}
```

That looks good. So we can run our migration to get that structure into our database.

From within the **Vagrant Machine** run:

```
> php artisan migrate
Migrated: 2014_09_18_104806_create_contacts_table
```

You can confirm this by reviewing your database structure.

Seeding Your Database

Now that we have a database we can put some seed (fake) data into our database to have something to work with.

Again we are going to use the generators package to help us with this task.

```
> php artisan generate:seed contacts
```

The last statement can be run from either your local machine or from vagrant. But it produces the following code inside `app/database/seeds/ContactsTableSeeder.php`:

```
<?php

// Composer: "fzaninotto/faker": "v1.3.0"
use Faker\Factory as Faker;

class ContactsTableSeeder extends Seeder {

    public function run()
    {
        $faker = Faker::create();

        foreach(range(1, 10) as $index)
        {
            Contact::create([

            ]);
        }
    }

}
```

From this we see that we are using the Faker package that we included earlier in our **composer.json** file. Also this Seeder requires that we have a model called **Contact**. So we can use the generator package to create a basic **Contact** model for us and then revisit that model at a later time.

Let's create that model:

```
> php artisan generate:model Contact
```

Now that we have a model, we can start using the seeder. Inside the `run` method we will add the following:

```
public function run()
{
    // clear out the database on subsequent seeding attempts
    DB::table('contacts')->truncate();

    $faker = Faker::create();

    // add a known record
    Contact::create([
        'firstName' => 'Eric',
        'lastName' => 'Jones',
        'birthday' => date('Y-m-d'),
        'street1' => '1 Monument Circle',
        'city' => 'Indianapolis',
        'state' => 'IN',
        'zip' => '46204',
        'email' => 'eric@example.com',
        'phone' => '(555) 123-4444'
    ]);

    // add 99 random records
    foreach(range(1, 99) as $index)
    {
        Contact::create([
            'firstName' => $faker->firstName,
            'lastName' => $faker->lastName,
            'birthday' => $faker->date,
            'street1' => $faker->streetAddress,
            'street2' => $faker->secondaryAddress,
            'city' => $faker->city,
            'state' => $faker->state,
            'zip' => $faker->postcode,
            'email' => $faker->email,
            'phone' => $faker->phoneNumber
        ]);
    }
}
```

Once we have added completed our seed file, we need to tell Laravel about it. Within the file `app/database/seed/DatabaseSeeder.php` add the following to the `run` method:

```
public function run()
{
    Eloquent::unguard();
    $this->call('ContactsTableSeeder.php');
}
```

Now that Laravel knows about our seed file, we can seed our database.

The following command needs to be run from within **Vagrant**:

```
> php artisan db:seed
Seeded: ContactsTableSeeder
```

Once the command has completed nothing spectacular happens, so let's check our database to see if it was properly seeded. You should now see 100 records in your contacts table.

Contacts API

Our next step in building our application is creating our contacts RESTful API. Laravel comes packaged with a RESTful-like interface so we will be leveraging what Laravel has provided.

Routing

Open the file `app/routes.php`. You will see a single base route in this file, we will leave that for now. However, under that route create a following **Resource** route:

```
...

Route::resource('contact', 'ContactController', [
    'except' => ['create', 'edit']
]);
```

Since the routes for **create**, and **edit** provide forms to create and edit a resource, we don't want Laravel to respond to those requests since the implementor of the API should provide the forms to created and edit a contact. We can see what routes this provides by running the command

```
> php artisan routes
```

We can see that we have:

- contact.index (GET) which provides all contacts
- contact.store (POST) which creates a new contact
- contact.show (GET) which provides a single contact
- contact.update (PUT/PATCH) which updates a single contact
- contact.delete (DELETE) which deletes a single contact

Controller

The previously defined route

```
Route::resource('contact', 'ContactController', [
    'except' => ['create', 'edit']
]);
```

you will notice that the route is expecting the **ContactController**.

Let's create that now:

```
> php artisan controller:make ContactController --except=create,edit
Controller created successfully!
```

After the controller has been made open file `app/controllers/ContactController.php`. You should see the class ContactController with a few empty methods.

ContactController Index

We will first complete the index method of this controller. The index method should provide all contacts, and since this going to be an API we should return the contacts list as a JSON string.

```
public function index()
{
    $contacts = Contact::all();
    return Response::json($contacts);
}
```

To test APIs like this I prefer too use a Google Chrome app called 'Postman - REST Client'. There are two versions of this app, and I prefer the non-packaged app. This app allows me to test the various requests (GET, POST, PUT, DELETE), and easily see the results from the server.

Open **Postman**, and where it says, "Enter request URL Here", we will put the following URL

```
http://contactmgr.app:8000/contact
```

Make sure that the request type is "GET", and press the blue "Send" button. Once the request is complete you should now see all the contacts within a JSON string.

ContactController Store

The store method is for creating a new contact record. So we need to

1. Validate the incoming data
2. Create a new contact record
3. Return a response

We could do all that work within this method; however, a better practice to move a lot of that work out side of the controller into other classes.

And that's what we will do.

Within the folder `app/IndyDev` create two folders

- Repository
- Validator

The `Repository` folder will contain a classes that house a lot of the business logic of creating, updating, and building contact records.

The `Validator` folder will contain classes that validate data when creating and updating a contact record.

Validator

Let's start with the validator by creating two files within this folder.

- `app/Validator/Validator.php`
- `app/Validator/ContactValidator.php`

Within the `app/Validator/Validator.php` class, add the following:

```php
<?php

namespace IndyDev\Validator;

use Validator as V;

/**
 * Validator
 *
 * An abstract class to validate a data set.
 */
abstract class Validator {

    /**
     * Validation Errors
     *
     * @var Illuminate\Support\MessageBag
     */
    protected $errors;

    /**
     * Is Valid
     *
     * Validates the provided data against a set of validation
     * rules.  Return true if data is valid, false if data
     * is invalid.
     *
     * @param array $data The data to be validated.
     *
     * @return boolean
     */
    public function isValid(array $data)
    {
        // makes the Laravel Validator class
        $validator = V::make($data, static::$rules);

        // if validation fails, store the errors and return false
        if ($validator->fails()) {
            $this->errors = $validator->messages();
            return false;
        }

        // the data passed validation
        return true;
    }

    /**
     * Get Errors
     *
     * Provides the errors generated when validating data.
     *
     * @return Illuminate\Support\MessageBag
     */
    public function getErrors()
    {
        return $this->errors;
    }
}
```

This abstract class (i.e., a class that is requiried to be extended) provides some basic validation processes. If you wanted to take this class further, you could create additional methods to check if the incoming data is valid for creation or if data is valid for update. But I will leave that exercise for you at a later time.

Since this class is required to be extended let's extend it within our `app/Validator/ContactValidator.php` class.

```php
<?php

namespace IndyDev\Validator;

/**
 * Contact Validator
 *
 * Validates contact record data
 */
class ContactValidator extends Validator {

    /**
     * Contact Validation Rules
     *
     * @var array
     */
    protected static $rules = [
        'firstName' => 'required|max:255',
        'lastName'  => 'max:255',
        'birthday'  => 'date',
        'email'     => 'email|max:255',
        'phone'     => 'max:255',
        'street1'   => 'max:255',
        'street2'   => 'max:255',
        'city'      => 'max:255',
        'state'     => 'max:255',
        'zip'       => 'max:255'
    ];
}
```

Since this class extends our Validator class we inherit the functionality from that class (e.g., the method is valid). If this application were to ever grow, all we would have to do is create another class very similar to this one.

Hopefully, you can see how this is more beneficial than having to recreate the validation process in every controller multiple times.

Respository

There is a lot going on within the ContactRepository class, so we will review each method of the class individually and than as a whole.

```php
<?php

namespace IndyDev\Repository;

use Contact;
use IndyDev\Validator\ContactValidator;

class ContactRepository
{
    protected $validator;

    public function __construct(ContactValidator $validator)
    {
        $this->validator = $validator;
    }

}
```

This is the constructor which requires that we set the ContactValidator class. A thing to note is that we won't have to explicitly provide the validator for this class as Laravel will find our validator class and provide it to our repository automatically.

ContactRepository::all();

```php
public function all()
{
    return Contact::orderBy('lastName', 'asc')->get();
}
```

The all method returns all contacts ordered by last name.

ContactRepository::get()

```
public function get($contact_id)
{
    return Contact::findOrFail((int) $contact_id);
}
```

The get method simply gets a single contact record or fails. If a failure occurs this method will throw a **ModelNotFoundException** which can then be handled appropriately.

Model Not Found Exception

A simple way to handle the **ModelNotFoundException** is to update the file `app/start/global.php`. Update this file to include the following:

```
<?php

use Illuminate\Database\Eloquent\ModelNotFoundException;

...

// beneath the existing App::error()

App::error(function(ModelNotFoundException $exception)
{
    return Response::make('Page Not Found', 404);
});
```

Now any time our application cannot find the requested model, our application will return a 404 Page Not Found error. Note: this method can be extended to include views and additional logic, but I will leave that as a self-exercise.

ContactRepository::create()

```
public function create(array $data)
{
    if ($this->validator->isValid($data)) {

        return Contact::create([
            'firstName' => array_get($data, 'firstName'),
            'lastName'  => array_get($data, 'lastName'),
            'birthday'  => array_get($data, 'birthday'),
            'email'     => array_get($data, 'email'),
            'phone'     => array_get($data, 'phone'),
            'street1'   => array_get($data, 'street1'),
            'street2'   => array_get($data, 'street2'),
            'city'      => array_get($data, 'city'),
            'state'     => array_get($data, 'state'),
            'zip'       => array_get    ($data, 'zip')
        ]);

    } else {
        return false;
    }
}
```

This is the create method which will create and then return a new contact record. Note that this method uses the ContactValidator::isValid method to validate the incoming data before it creates the record. If the data is invalid the method returns false.

ContactRepository::update()

```
public function update($contact_id, array $data)
{
    if ($this->validator->isValid($data)) {

        $contact = $this->get($contact_id);

        $contact->firstName = array_get($data, 'firstName');

        $contact->lastName  = array_get($data, 'lastName', $contact->lastName);
        $contact->birthday  = array_get($data, 'birthday', $contact->birthday);
        $contact->email     = array_get($data, 'email',    $contact->email);
        $contact->phone     = array_get($data, 'phone',    $contact->phone);
        $contact->street1   = array_get($data, 'street1',  $contact->street1);
        $contact->street2   = array_get($data, 'street2',  $contact->street2);
        $contact->city      = array_get($data, 'city',     $contact->city);
        $contact->state     = array_get($data, 'state',    $contact->state);
        $contact->zip       = array_get($data, 'zip',      $contact->zip);

        $contact->save();

        return $contact;

    } else {
        return false;
    }
}
```

This is the update method, it accepts the contact id as the first parameter and the updated contact data as the second parameter. The method validates the incoming data, gets the orignal contact record, updates the contact record with the new data, saves the contact and then returns the newly updated contact. If the incoming data is invalid the method returns false.

ContactRepository::error()

```
public function errors()
{
    return $this->validator->getErrors();
}
```

For both the create and update methods, if validation fails a false is returned. We need to provide a way in which the calling code can access the validation errors. So we created the errors method which will provide any errors generated during the validation process.

ContactRespository::delete()

```
public function delete($contact_id)
{
    $contact = $this->get($contact_id);
    $contact->delete();
}
```

And finally, this is the delete method which accepts the contact id as the only parameter.

Updating the Controller

Now that we have an abstracted contact responsitory and validation, lets update the controller to reflect these new changes.

At the top of the file include:

```
use IndyDev\Repository\ContactRepository;
```

After which update the constructor to accept our repository as a parameter:

```
protected $contact;

public function __construct(ContactRepository $contact)
{
    $this->contact = $contact;
}
```

Within your POSTMAN client, make a new request and make sure that nothing is failing. If so identify the cause and fix it. But hopefully, you shouldn't have any issues.

Now let's update the index method to be as follows:

```
public function index()
{
    $contacts = $this->contact->all();
    return Response::json($contacts)
}
```

Are there any issues?

ContactController Store Method

If not let's proceed with the store method. And since we abstracted a lot of our business logic and validation into the ContactRespository class our store method is pretty simple.

```
public function store()
{
    $contact = $this->contact->create(Input::all());
    return ($contact)
        ? Response::json($contact, 201)
        : Response::json($this->contact->errors(), 400);
}
```

Let's try this out. Within Postman, adjust the request to be a post request, and test the validation, and then test creating a new record with proper data.

You probably have gotten an error. Something to the effect of a MassAssignmentException if everything is working correctly.

Updating The Contact Model

So let's fix this mass assignment exception, open up `app/models/Contact.php`. Right now our model is pretty much empty and should resemble this.

```
<?php

class Contact extends \Eloquent {
    protected $fillable = [];
}
```

The attribute **$fillable** is what is causing the error. Fillable is a special attribute as it tells Laravel that only the contact attributes within this array can be mass assigned. Laravel provides the inverse of this method also called **guarded**. The **guarded** array indicates which contact attributes cannot be mass assgined.

Let's update our **ContactModel** to reflect the following changes:

```
<?php

class Contact extends Eloquent
{
    protected $table = 'contacts';
    protected $guarded = ['id', 'created_at', 'updated_at'];
}
```

with these changes we are saying allow the mass assignment of all contact attributes except **id**, **created_at**, and **updated_at**. So let's try and create another contact record. You should have now received the JSON string for the new contact record.

ContactController Update Method

With the store method out of the way, we can work on the update method which is very similar to the store method

```
public function update($id)
{
    $contact = $this->contact->update($id, Input::all());
    return ($contact)
        ? Response::json($contact, 200)
        : Response::json($this->contact->errors()->all(), 400);
}
```

The only differing portions are we are:

- using the ContactRespository::update() method,
- returning a status code of 200 instead of 201.

With the newly created record from the previous testing, change a value and see if the contact record is changed.

Note: with put requests, you will need to send your data **x-www-form-urlencoded**

Any issues?

ContactController Show Method

With the contact controller show method, we will simply return the JSON encoded contact record.

```
public function show($id)
{
    $contact = $this->contact->get($id);
    return Response::json($contact);
}
```

Test this witin postman, and you should receive the contact record. If you test it with an invalid record number you should get a 404 Page Not Found response.

Before moving on let's add a dynamic attribute to our model. Laravel allows us to add a dynamic attribute that should be included when the model is converted to a JSON string.

```
<?php

use Carbon\Carbon;

class Contact extends Eloquent
{
    ...

    protected $appends = ['is_birthday'];

    public function getIsBirthdayAttribute()
    {
        if (empty($this->attributes['birthday'])) return false;

        $birthday = new Carbon($this->attributes['birthday']);

        return ($birthday->month == date('m') && $birthday->day == date('d'));
    }
}
```

This will add the 'is_birthday' boolean attribute.

ContactController Destroy Method

Our final method to complete within the contact controller is the destroy method. Which looks like this.

```
public function destroy($id)
{
    $this->contact->delete($id);
    return Response::json('Deleted', 200);
}
```

The response simply indicates that the contact record has been deleted.

API Finished

Our simple api is now finished, we can view all contact records, view a single contact record, create a new contact record, update an existing contact record, and delete an existing contact record.

There are things which can be done to extend this api and we will brainstorm some of those ideas at the end of the session.

But the API is only the first half of our application. We still need to build out the front-end of our application using AngularJS.

Building Out AngularJS

We want it so that anyone who hits the page `http://contactmgr.app:8000` to be shown a list of contacts within the contact manager. This index page will be the page that "contains" our AngularJS app but be loaded by Laravel routing.

There are a few different ways to load the AngluarJS app, and this is just one of the few options that you have.

Open the file `app/routes.php`

```
Route::get('/', function()
{
    return View::make('hello');
});

Route::resource('contact', 'ContactController', [
    'except' => ['create', 'edit']
]);
```

Laravel comes default with a route to handle the index page we need; however, lets rename the file to something like "main" instead of hello.

Open the `app/views/hello.php` and update the file as follows:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Contact Manager Application</title>

</head>
<body>

</body>
</html>
```

and save the file as `app/views/main.php`.

AngularJS / Bootstrap Setup

Before we include AngluarJS and Bootstrap CSS into our project let's create the directory structure required.

Within the public folder create the following directory sturcture:

- public
    - css
    - js
        - a
            - controllers
            - filters
            - ngmodules
            - views

AngularJS

To include AngularJS into our project:

1. Navigate to the web page `https://code.angularjs.org/1.2.19/`
2. Download the files:
    - **angular.js**
    - **angular-resourse.js**
    - **angular-route.js**
3. Move the **angular.js** file into `public/js/a` folder
4. Move the **angular-resourse.js** and **angular-route.js** files into the `public/js/a/ngmodules` folder

Bootstrap

Navigate to `getbootstrap.com` and click the **Download** button. We will only be incluing the Bootstrap CSS from the CDN. If network speed becomes a problem we can download the bootstrap CSS files otherwise we will simply use the CDN.

Create a new file `public/css/styles.css` with the following content:

```
.navbar-center {
    position: absolute;
    width: 100%;
    height: 100%;
    top: 0;
    left: 0;
    text-align: center;
    padding-top: 15px;
    padding-bottom: 15px;
}

.navbar-center .navbar-brand { float: none; line-height: 25px; }

@media (min-width: 768px) {
    .navbar-center .navbar-brand { line-height: 20px; }
}
```

Updating main.php

With these new files let's update our main.php to include these files.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
    <meta charset="UTF-8">
    <title>Contact Manager Application</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet" type="text/css" href="/css/style.css">

    <script type="text/javascript" src="/js/a/angular.js"></script>
    <script type="text/javascript" src="/js/a/ngmodules/angular-route.js"></script>
    <script type="text/javascript" src="/js/a/ngmodules/angular-resource.js"></script>
</head>
<body>

    <button class="btn btn-primary">{{ 'Hello AngluarJS' }}</button>

</body>
</html>
```

With this we can test if we have AngularJS and Bootstrap being brought in correctly. If you don't have this displaying correctly, review your Javascript console and debug.

Creating Our ContactMgrApp

Create a new file `public/js/a/ContactMgr.js` and start the file with this code

```
angular.module('ContactMgrApp', ['ngRoute', 'ngResource'])
```

This will initialize our application and indicate that our application depends on the **ngRoute** and **ngResourse** modules.

Update our `app/views/main.php` file to be the following:

```
<!DOCTYPE html>
<html lang="en" ng-app="ContactMgrApp">
<head>

    ...

    <script type="text/javascript" src="/js/a/ContactMgr.js"></script>

</head>
<body>

    <button class="btn btn-primary">{{ 'Hello AngluarJS' }}</button>

</body>
</html>
```

Creating The ContactMgr Controllers

Our next step is to start creating some AngularJS controllers by first creating the file `public/js/a/controllers/ContactControllers.js` and adding the following:

```
angular.module('ContactMgrApp')

.controller('ContactsCtrl', ['$scope', function ($scope) {

    $scope.hello = "Hello World!";

}])
```

Once you have this update the file `app/views/main.php`:

```
...

<script type="text/javascript" src="/js/a/controllers/ContactController.js"></script>

</head>
<body ng-controller="ContactsCtrl">

    <button class="btn btn-primary">{{ 'Hello AngluarJS' }}</button>

    <h1>{{ hello }}</h1>

</body>

...
```

Load this page into your browser and you should see that the H1 tag now has "Hello, World!". AngularJS is a very complex and exciting technology so in order for me to complete this workshop I won't cover all of the different aspects of AngularJS. But I do hope that this workshop peaks your interest in both Laravel and AngularJS, and motivates you to find out more.

With that let's jump right into the more complex features of this application.

AngulaJS ContactCtrl (Viewing All Contacts)

Open the files (if they are not already open)

- `public/js/a/ContactMgr.js`
- `public/js/a/controllers/ContactControllers.js`

Within the ContactMgrApp we will add a constant for our application that points to our contact data previously built.

```
angular.module('ContactMgrApp', ['ngRoute', 'ngResource'])

.constant('contactsDataUrl', 'http://contactmgr.app:8000/contact/')
```

Now update the controller `ContactsCtrl` with the following:

```
...

.controller('ContactsCtrl', ['$scope', '$http', 'contactsDataUrl', function ($scope, $http, contactsDataUrl) {

    $scope.data = {};

    $http.get(contactsDataUrl)
        .success(function (contacts) {
            $scope.data.contacts = contacts;
        })
        .error(function (response) {
            $scope.data.error = response;
        });

}])
```

with this code we are injecting the $http service and our constant. I create a scope variable data and set it to an empty object. I am doing this with the data object because it becomes easier to use and extend later on down the road.

The next lines use the $http service to make a call to the server. If the response was a success, set the contacts attribute on the data object; otherwise, get the response and set the error attribute on the data object.

Finally, within the `app/views/main.php` file update the HTML code which displays the results

```
<div class="container-fluid">

    <div ng-show="data.error">
        <h1>Server Error</h1>
        <div class="alert alert-danger">
            {{ data.error }}
        </div>
    </div>

    <div ng-hide="data.error">
        <h1>Contacts</h1>
        <div class="list-group">

            <a ng-repeat="contact in data.contacts" class="list-group-item">
                {{ contact.firstName }} {{ contact.lastName }}
            </a>

        </div>
    </div>
</div>
```

If we refresh the page we should hopefully see a listing of contacts from our server. If not debug. We can see that since there was no error, the second block is displayed. Within the second block we repeat over the contact listing displaying each one.

To test the error message, witin `app/controllers/ContactConroller.php` change the response in **index()** to something similar to

```
public function index()
{
    return Response::json('Danger Danger Danger', 500);
    $contacts = $this->contact->all();
    return Response::json($contacts);
}
```

Which will trigger the error to be displayed. Once you have see that the error message works remove the 500 response line.

AngularJS Resources

Before we move any further, we need to talk about ngResource module and the $resource service. We can access our code via the $http service, however, using a RESTful api like we are is so common-place the AngularJS team provided to us a way to short-cut the $http code.

By using this resource we save on code and gain the following methods.

```
{ 'get':    {method:'GET'},
'save':   {method:'POST'},
'query':  {method:'GET', isArray:true},
'remove': {method:'DELETE'},
'delete': {method:'DELETE'} };
```

Within the `public/js/a/ContactMgr.js` file add the following:

```
...

.factory('Contact', ['$resource', 'contactsDataUrl', function ($resource, contactsDataUrl) {
    return $resource(
        contactsDataUrl + ':id',
        null,
        {
            'update': { method: 'PUT' }
        }
    );
}]);
```

With this code we are creating a service called 'Contact'. Contact uses the $resource service to access the server in a RESTful manner. The first parameter is how the resource should access our data. Notice that we are adding in the **:id** route parameter, if that parameter is empty it will access the url `contact/` else if it is not empty `contact/1`.

The second parameter is used to add default URL parameters, of which we have none so set it to null.

Lastly, the third parameter add a new method 'update' which uses the HTTP Method 'PUT' so send update requests.

Now let's remove the $http service from our `app/js/a/controllers/ContactController.js` file.

```
.controller('ContactsCtrl', ['$scope', 'Contact', function ($scope, Contact) {

    $scope.data = {};

    Contact.query(function (contacts) {
        $scope.data.contacts = contacts;
    }, function (response) {
        $scope.data.error = response.data;
    });

}])
```

Notice that we changed the parameters that get passed into the controller. We are using the **.query** method to access the `contact/` url and return an array of objects.

If we refresh our application's page we should see no changes.

AngularJS Creating A New Contact

It is all well and good to view a listing of contacts witin your database, however, we want to add a new contact to our application. How should we go about doing that. We need to create a new angular controller that help us with this endeavor.

But before we create a new controller, we need to discuss and implement angular routing and views within our application.

When using ngRoute within your application, your are given a directive (**ng-view**), a provider (**$routeProvider**), and two services (**$route**, and **$routeParam**). We will be using all directly but **$route** in this application.

Routing

For this application I defined my angular routing witin the file `public/js/a/ContactMgr.js` so open that file and add the following:

```
.config(['$routeProvider', function ($routeProvider) {

    $routeProvider.when('/contact/create', {
        templateUrl: "js/a/views/contactCreate.html",
        controller: 'contactCreateCtrl'
    });

    $routeProvider.otherwise({
        templateUrl: "js/a/views/contactMain.html"
    });
}])
```

Coming from Laravel Routing this may seem oddly familiar but also foreign at the same time. **$routeProvider** allows us to define which controller should be used and which view should be loaded when the appliation is at the defined URL. For example when the application is at the URL `/contact/create`, Angular should load the view `js/a/views/contactCreate.html`, and the controller `contactCreateCtrl`.

I have included as a catch all route **otherwise** to view all contact listings.

Create these new files within the `js/a/views/` folder

- contactMain.html
- contactCreate.hml
- contactForm.html

Within the `contactMain.html` add the following:

```
<div class="navbar navbar-default navbar-static-top" role="navigation">
    <div class="container-fluid">
        <p class="navbar-center"><a class="navbar-brand" href="#">Rolodex</a></p>
        <ul class="nav navbar-nav pull-right">
            <li><a href="#/contact/create">Create</a></li>
        </ul>
    </div>
</div>

<div class="container-fluid">

    <div class="list-group">
        <a ng-repeat="contact in data.contacts | orderBy: 'lastName' "
            class="list-group-item">
            {{ contact.firstName }} {{ contact.lastName }}
        </a>
    </div>
</div>
```

This is very similar to what we had within the `app/views/Main.php` a moment ago. However, I have added a navigation bar and added a filter to our ng-repeat directory. The filter **orderBy** will order our contact by lastName if it wasn't already so.

Within the `contactCreate.html` we need to add the following:

```
<div class="navbar navbar-default navbar-static-top" role="navigation">
    <div class="container-fluid">
        <p class="navbar-center"><a class="navbar-brand" href="#">Rolodex</a></p>
        <ul class="nav navbar-nav pull-left">
            <li><a href="/">Back</a></li>
        </ul>
    </div>
</div>
<div class="container-fluid">
    <div id="contact-create" ng-controller="contactCreateCtrl">

        <div class="page-header"><h1>Create New Contact</h1></div>

        <div class="alert alert-danger" ng-show="validation">
            <strong>Validation Errors</strong>
            <p ng-repeat="err in validation">{{ err }}<p>
        </div>


        <form id="contact-create-form" class="form-horizontal" ng-submit="addContact()"
              ng-include="'js/a/views/contactForm.html'">
        </form>

    </div>

</div>
```

Within this view we have a navigation bar similar to our `contactMain.html` file. We use the **ng-controller** directive to set the controller scope for the code within that block. We have a block that shows if we have some validation issues. And finally, we have a form element that has the **ng-submit** and **ng-include** directives. The **addContact()** method for the **ng-submit** needs to be defined within our **contactCreateCtrl**. We also need to create the file `js/a/views/contactForm.html` from our **ng-include** directive.

Let's create the `contactForm.html` file now.

```
<fieldset>
    <legend>Basic</legend>
    <div class="form-group">
        <label for="contact-firstName" class="control-label col-sm-2">First Name</label>
        <div class="col-sm-10">
            <input id="contact-firstName" class="form-control" ng-model="contact.firstName">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-lastName" class="control-label col-sm-2">Last Name</label>
        <div class="col-sm-10">
            <input id="contact-lastName" class="form-control" ng-model="contact.lastName">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-email" class="control-label col-sm-2">Email</label>
        <div class="col-sm-10">
            <input type="email" id="contact-email" class="form-control" ng-model="contact.email">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-phone" class="control-label col-sm-2">Phone</label>
        <div class="col-sm-10">
            <input type="phone" id="contact-phone" class="form-control" ng-model="contact.phone">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-birthday" class="control-label col-sm-2">Birthday</label>
        <div class="col-sm-10">
            <input id="contact-birthday" class="form-control" ng-model="contact.birthday" type="date">
        </div>
    </div>
</fieldset>
<fieldset>
```

```
    <legend>Address</legend>
    <div class="form-group">
        <label for="contact-street1" class="control-label col-sm-2">Street 1</label>
        <div class="col-sm-10">
            <input id="contact-street1" class="form-control" ng-model="contact.street1">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-street2" class="control-label col-sm-2">Street2</label>
        <div class="col-sm-10">
            <input id="contact-street2" class="form-control" ng-model="contact.street2">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-city" class="control-label col-sm-2">City</label>
        <div class="col-sm-10">
            <input id="contact-city" class="form-control" ng-model="contact.city">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-state" class="control-label col-sm-2">State</label>
        <div class="col-sm-10">
            <input id="contact-state" class="form-control" ng-model="contact.state">
        </div>
    </div>
    <div class="form-group">
        <label for="contact-zip" class="control-label col-sm-2">Zip Code</label>
        <div class="col-sm-10">
            <input id="contact-zip" class="form-control" ng-model="contact.zip">
        </div>
    </div>
</fieldset>
<div class="form-group form-actions">
    <div class="col-sm-offset-2 col-sm-10">
        <button type="submit" class="btn btn-primary">Save</button>
        <a href="/" class="btn btn-link">Cancel</a>
    </div>
</div>
```

Feel free to not to include all of the inputs; however, the first name input is required.

Open the file `public/js/a/controllers/ContactController.js` and add the `contactCreateCtrl`.

```
...

.controller('contactCreateCtrl', ['$scope', '$location', 'Contact', function ($scope, $location, Contact) {

    $scope.contact = {};

    $scope.addContact = function () {

        Contact.save($scope.contact, function (contact) {

            $scope.data.contacts.push(contact);
            $location.url('/');

        }, function (response) {

            $scope.validation = response.data;

        });

    };

}])
```

Update the file `app/views/main.php` with this code:

```
<div ng-show="data.error">
    <h1>Server Error</h1>
    <div class="alert alert-danger">
        {{ data.error }}
    </div>
</div>

<div ng-hide="data.error"><ng-view /></div>
```

The first block of code should be familiar to you the second block has been updated with the directive **ng-view**. If you recall, I stated the the **ngRoute** module provided the directive **ng-view**. The template defined within the angular routes will be place within this directive and the view updated.

Navigate to the base of your app `http://contactmgr.app:8000` and refresh the page. You should see a navigation bar at the top with a link to "Create" a new contact, and the listing of contacts below.

If you do not see this screen, check your javascript console and debug.

Click on the "Create" link and you should now see an updated nav bar and the form to create a new contact. Let's spend some time to test this out. First submit a form without any data. Does the application an error message? Create a new and valid contact. Does the application navigate back to the contact listing? Is your new contact within that listing? Refresh the page and is your contact still there?

Editing a Contact

Now that we have a way to create a contact how about let's create a way to edit a contact.

To do this we need to add the route, create the controller and view.

To add the route open the file `public/js/a/ContactMgr.js` and add the following before the **otherwise** route:

```
...

$routeProvider.when('/contact/:contactId/edit', {
    templateUrl: "js/a/views/contactEdit.html",
    controller: 'contactEditCtrl'
});

...
```

This is very similar to the create route however, we have a route parameter **:contactId** (which is very similar to how Laravel does its variable routing).

To add the controller open and edit the file `public/js/a/controllers/ContactController.js` and add the following to this file:

```
...

.controller('contactEditCtrl', ['$scope', '$location', '$routeParams', 'Contact', function($scope, $location, $routeParams,
Contact) {

    var contactId = $routeParams.contactId;

    Contact.get({ id: contactId }, function (contact) {
        $scope.contact = contact;
    }, function (response) {
        $scope.validation.error = response.data;
    });

    $scope.updateContact = function () {
        $scope.contact.$update({ id: contactId }, $scope.contact, function () {

            angular.forEach($scope.data.contacts, function (contact, i) {
                if (contact.id == contactId) {
                    $scope.data.contacts[i] = $scope.contact;
                }
            });

            $location.url('/');
        }, function (response) {
            $scope.validation = response.data;
        });
    };

}]);
```

With our controller in place, we can finish up the edit functionality by updating `contactMain.html` and creating our edit view.

Open the file `public/js/a/views/contactMain.html` and update the **ng-repeat** to include the **ng-click** directive for each list item.

```
<a ng-repeat="contact in data.contacts | orderBy: 'lastName' "
    class="list-group-item"
    ng-click="selectContact(contact)">
    {{ contact.firstName }} {{ contact.lastName }}
    <span class="pull-right"><i class="glyphicon glyphicon-chevron-right"></i></span>
</a>
```

Create the file `public/js/a/views/contactEdit.html` and add the following (or copy from the contactCreate.html and modify)

```
<div class="navbar navbar-default navbar-static-top" role="navigation">
    <div class="container-fluid">
        <p class="navbar-center"><a class="navbar-brand" href="#">Rolodex</a></p>
        <ul class="nav navbar-nav pull-left">
            <li><a href="/">Back</a></li>
        </ul>
    </div>
</div>
<div class="container-fluid">
    <div id="contact-edit" ng-controller="contactEditCtrl">

        <div class="page-header"><h1>Edit Contact {{ contact.firstName }}</h1></div>

        <div class="alert alert-danger" ng-show="validation">
            <strong>Validation Errors</strong>
            <p ng-repeat="err in validation">{{ err }}<p>
        </div>

        <form id="contact-edit-form" class="form-horizontal" ng-submit="updateContact()"
              ng-include="'js/a/views/contactForm.html'">
        </form>

    </div>
</div>
```

With this in place we should now have a way to edit a contact test it out. If you encounter any errors, now is the time to debug.

Contact Search

Having a long list of 100 contats can be tough to find a single individual. So to wrap up this application we will be adding a search functonality.

Create a new file `public/js/a/filters/customFilter.js`. We will create a new module from which we will add our new filter to be applied to our contact list.

```
angular.module('customFilters', [])

.filter('searchFullName', function () {

    return function (contacts, search) {

        if (angular.isArray(contacts) && angular.isString(search)) {

            var results = [],
                regExSearch = new RegExp ( search, 'i');

            for (var i = 0, c = contacts.length; i < c; i++) {

                var name = contacts[i].firstName + ' ' + contacts[i].lastName;

                if (regExSearch.test(name)) {

                    results.push(contacts[i]);

                }
            }

            return results;

        } else {

            return contacts;

        }
    }
});
```

To defined a filter, the first parameter must be the name of the filter, the second parameter is a factory function which returns a function that performs the filtering. The function returned by the factory defines two parameters, the first being the data that is required to be filtered (in our case it is the contact listing), and an additional parameter which I define as the search string.

This filter will loop through the contact list concatenating the contact's first and last name, and then compares the contacts full name against the given search string. If the contact's full name contains any portion of the search string, add that contact to the return array.

To add this module to our application update the `app/views/main.php` including the new module in our doucment head.

```
...

<script type="text/javascript" src="/js/a/filters/customFilters.js"></script>

....
```

Update the file `public/js/a/ContactMgr.js` by adding our **customFilters** module as a dependency.

```
angular.module('ContactMgrApp', ['ngRoute', 'ngResource', 'customFilters'])
```

After which we can now use the **searchFullName** filter within our `public/js/a/contactMain.html` file.

Add the form above the listing:

```
<form>
    <div class="form-group has-feedback">
        <label class="control-label">Search Contacts</label>
        <input type="search" class="form-control" ng-model="search">
        <span class="glyphicon glyphicon-search form-control-feedback"></span>
    </div>
</form>
```

Update the **ng-repeat** by adding the **searchFullName** filter to the end of the directive

```
...

<a ng-repeat="contact in data.contacts | orderBy: 'lastName' | searchFullName:search "

...
```

You should now be able to filter your contact list by a contact's first name, last name or a combination of the two.

Has Birthday

Wrapping Up

Thanks for spending this time with me going through this project. For the sake of brevity and simplier application development, certain concepts and functionality were either overlooked or removed. For example, a more realistic application would allow a contact to have multiple phone numbers, emails, and addresses. Also, an important concept in both Laravel and AngularJS is to provide test for your code which were not covered.

If you found this project interesting and exciting, I challenge you to explore these concepts more in depth and push the contact manager to a new level.

Thanks again.