Creating a Custom Event and Listener for Welcome Emails

In this example, I'll show you how to create a custom event and listener in Laravel to send a welcome email when a user registers. This follows the event-driven architecture that Laravel provides.

Step 1: Create the Event

First, let's create the event that will be dispatched when a user registers.

```php
// app/Events/UserRegistered.php

namespace App\Events;

use App\Models\User;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class UserRegistered
{
    use Dispatchable, SerializesModels;

    public $user;

    /**
     * Create a new event instance.
     *
     * @param User $user
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }
}
```

PROF

Step 2: Create the Listener

Next, create the listener that will handle sending the welcome email.

```php
// app/Listeners/SendWelcomeEmail.php

namespace App\Listeners;

use App\Events\UserRegistered;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
```

```php
use Illuminate\Support\Facades\Mail;
use App\Mail\WelcomeEmail;

class SendWelcomeEmail implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param UserRegistered $event
     * @return void
     */
    public function handle(UserRegistered $event)
    {
        Mail::to($event->user->email)
            ->send(new WelcomeEmail($event->user));
    }
}
```

Step 3: Create the Welcome Email

Create a mailable class for the welcome email.

```php
// app/Mail/WelcomeEmail.php

namespace App\Mail;

use App\Models\User;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class WelcomeEmail extends Mailable
{
    use Queueable, SerializesModels;

    public $user;

    /**
     * Create a new message instance.
     *
     * @param User $user
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
```

```
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->subject('Welcome to Our Application')
                    ->view('emails.welcome');
    }
}
```

Step 4: Create the Email Template

Create a Blade template for the email.

```
// resources/views/emails/welcome.blade.php

<!DOCTYPE html>
<html>
<head>
    <title>Welcome Email</title>
</head>
<body>
    <h1>Welcome, {{ $user->name }}!</h1>
    <p>Thank you for registering with our application.</p>
    <p>We're excited to have you on board!</p>
</body>
</html>
```

Step 5: Register the Event and Listener

Add the event and listener to the `EventServiceProvider`.

```
// app/Providers/EventServiceProvider.php

namespace App\Providers;

use App\Events\UserRegistered;
use App\Listeners\SendWelcomeEmail;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        UserRegistered::class => [
            SendWelcomeEmail::class,
        ],
```

```
    ];

    // ...
}
```

Step 6: Dispatch the Event

Finally, dispatch the event when a user registers. This would typically be in your registration controller.

```php
// app/Http/Controllers/Auth/RegisterController.php

namespace App\Http\Controllers\Auth;

use App\Events\UserRegistered;
use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;
use Illuminate\Foundation\Auth\RegistersUsers;

class RegisterController extends Controller
{
    use RegistersUsers;

    // ...

    protected function create(array $data)
    {
        $user = User::create([
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => Hash::make($data['password']),
        ]);

        // Dispatch the event
        event(new UserRegistered($user));

        return $user;
    }
}
```

PROF

Additional Notes

1. **Queueing**: The listener implements `ShouldQueue`, which means the email will be sent via a queue job. Make sure your queue worker is running.
2. **Testing**: You can test this by mocking the event or listener in your tests.

3. **Multiple Listeners**: You can add more listeners to the `UserRegistered` event if needed (e.g., for sending a Slack notification).
4. **Error Handling**: Consider adding error handling in the listener for cases where the email fails to send.

This implementation provides a clean separation of concerns and makes it easy to add additional actions when a user registers without modifying the registration logic itself.