Dave Liddament

Decoupling with Events

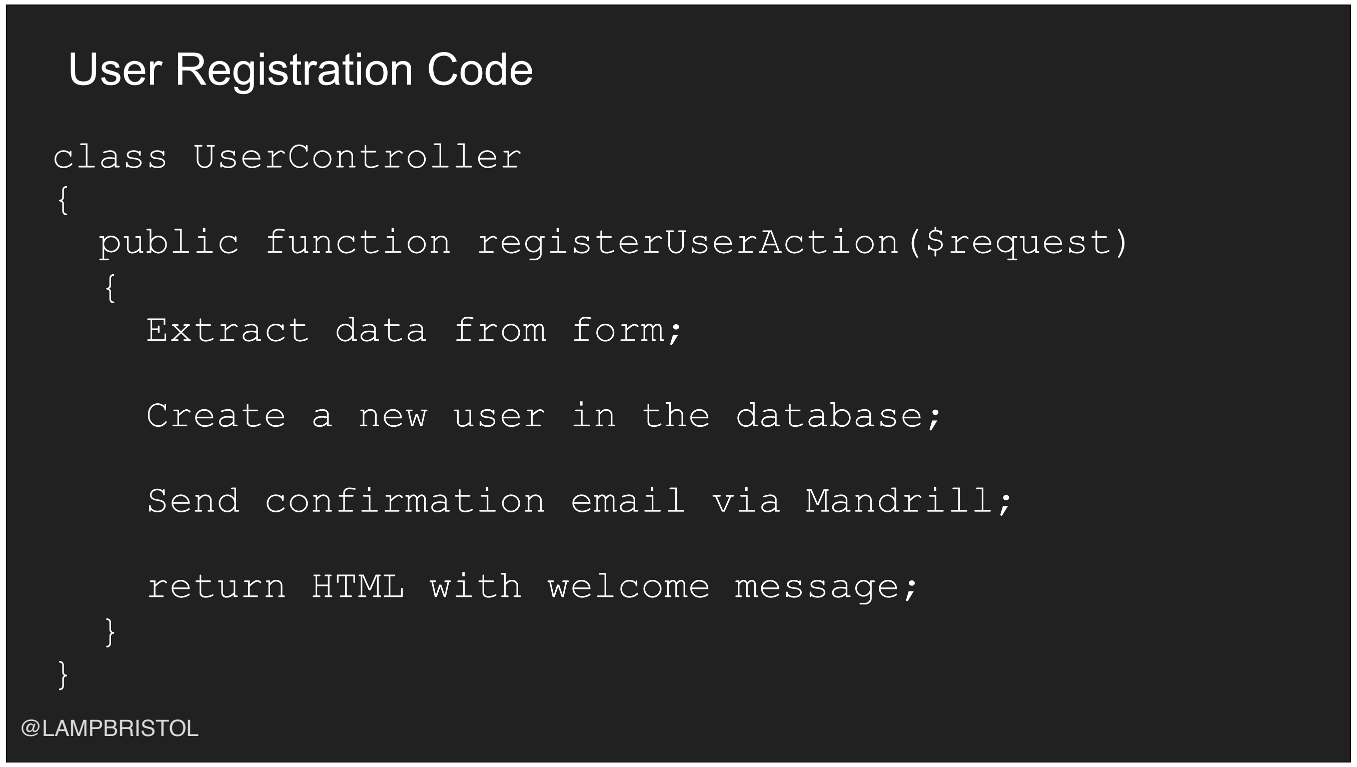
@phpsw WED 12TH October 2016

@LAMPBRISTOL

Why decoupling?

@LAMPBRISTOL

- Coupling refers to the degree of direct knowledge that one component has of another.
-Loose coupling or decoupling is reducing the that knowledge to the least possible .
-Loosely coupled code is easier to understand, test and extends.

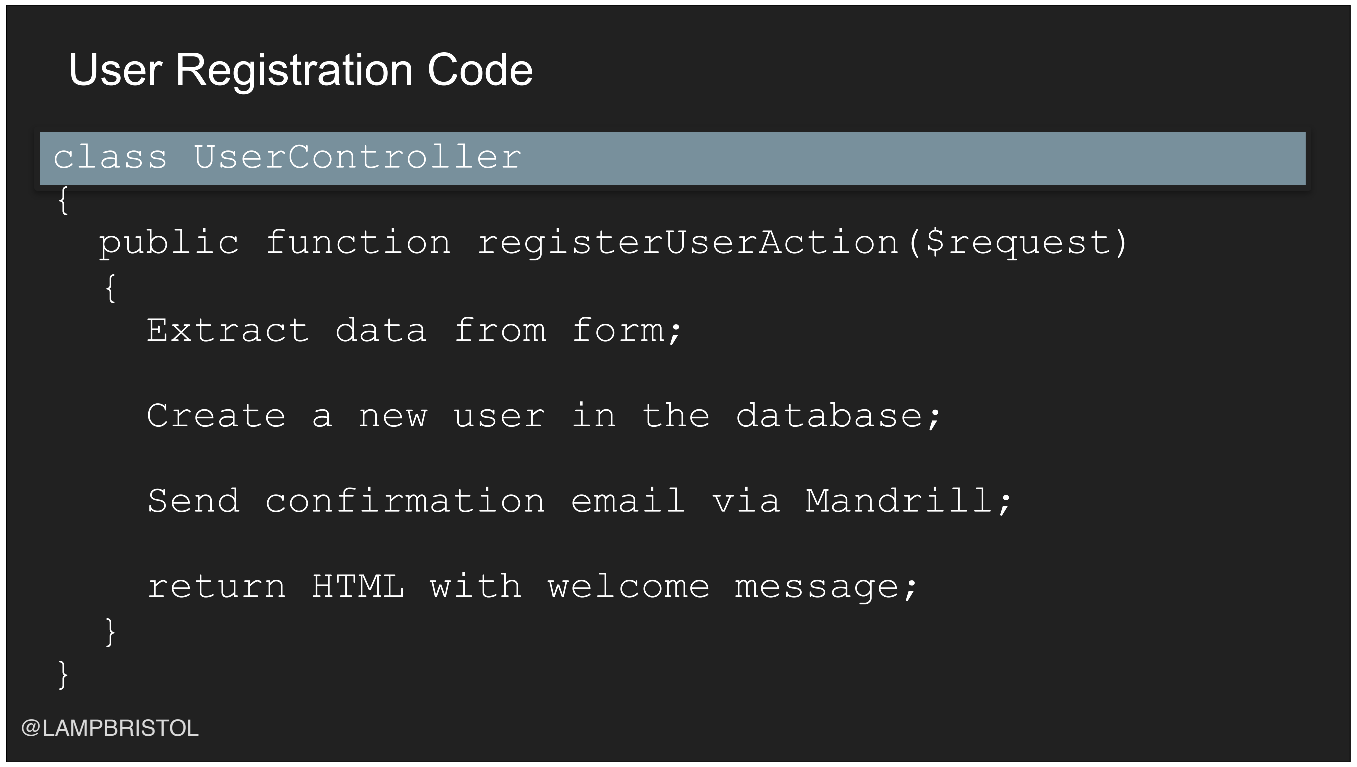


User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?

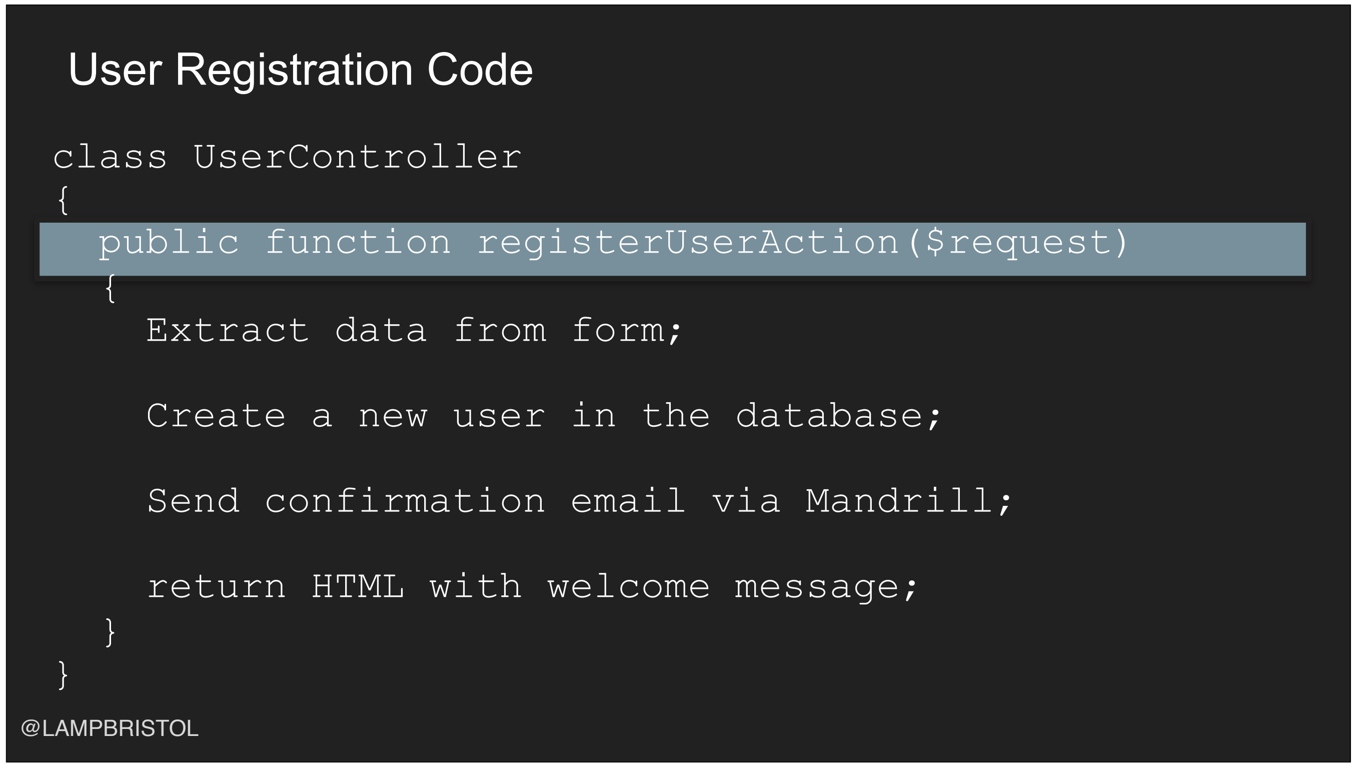


User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?

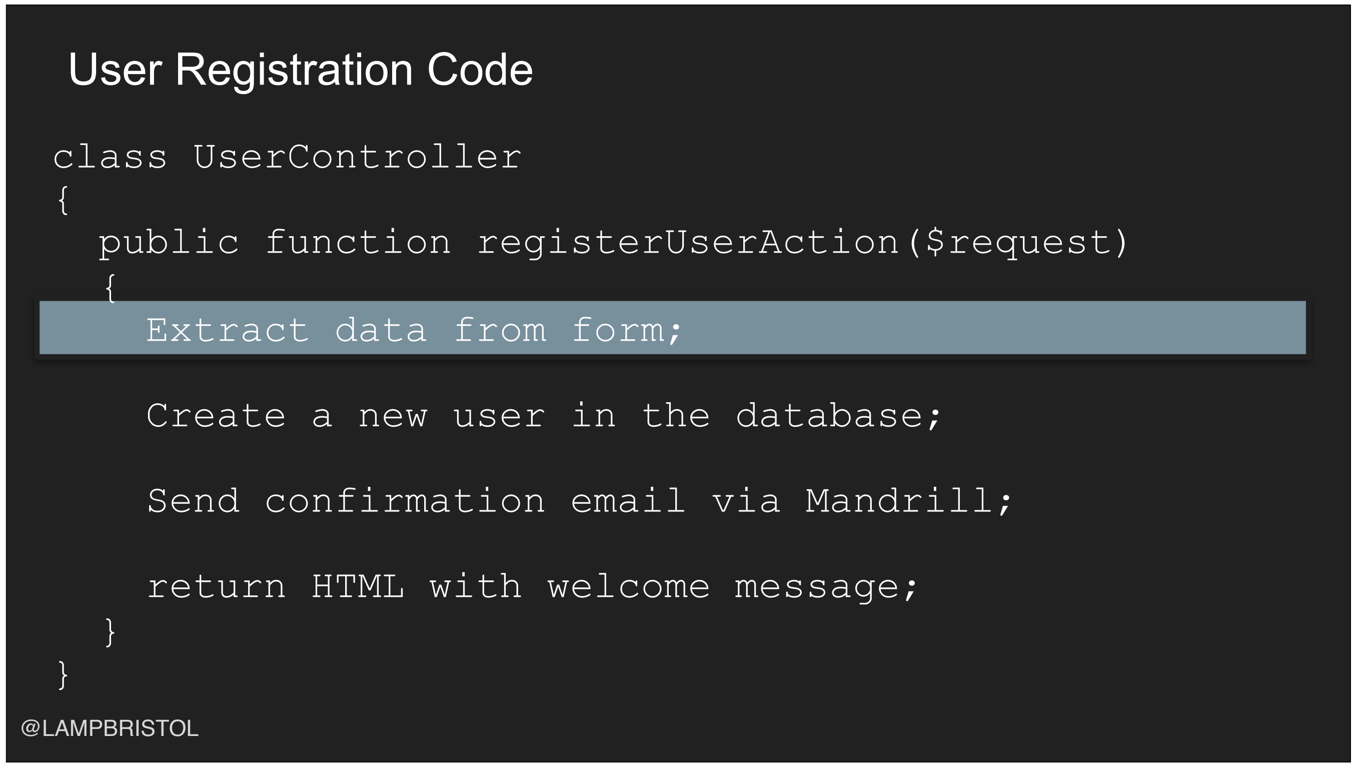


User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

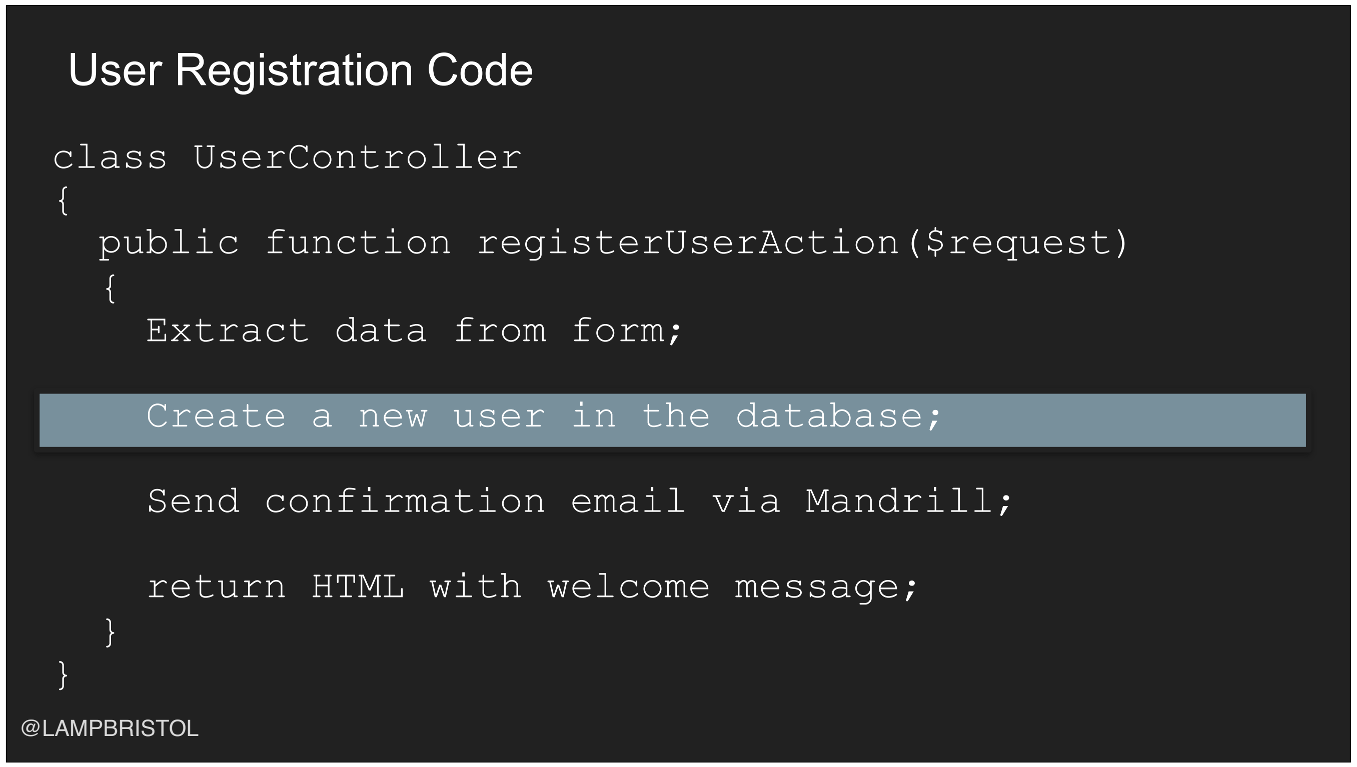
- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?



User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?



User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

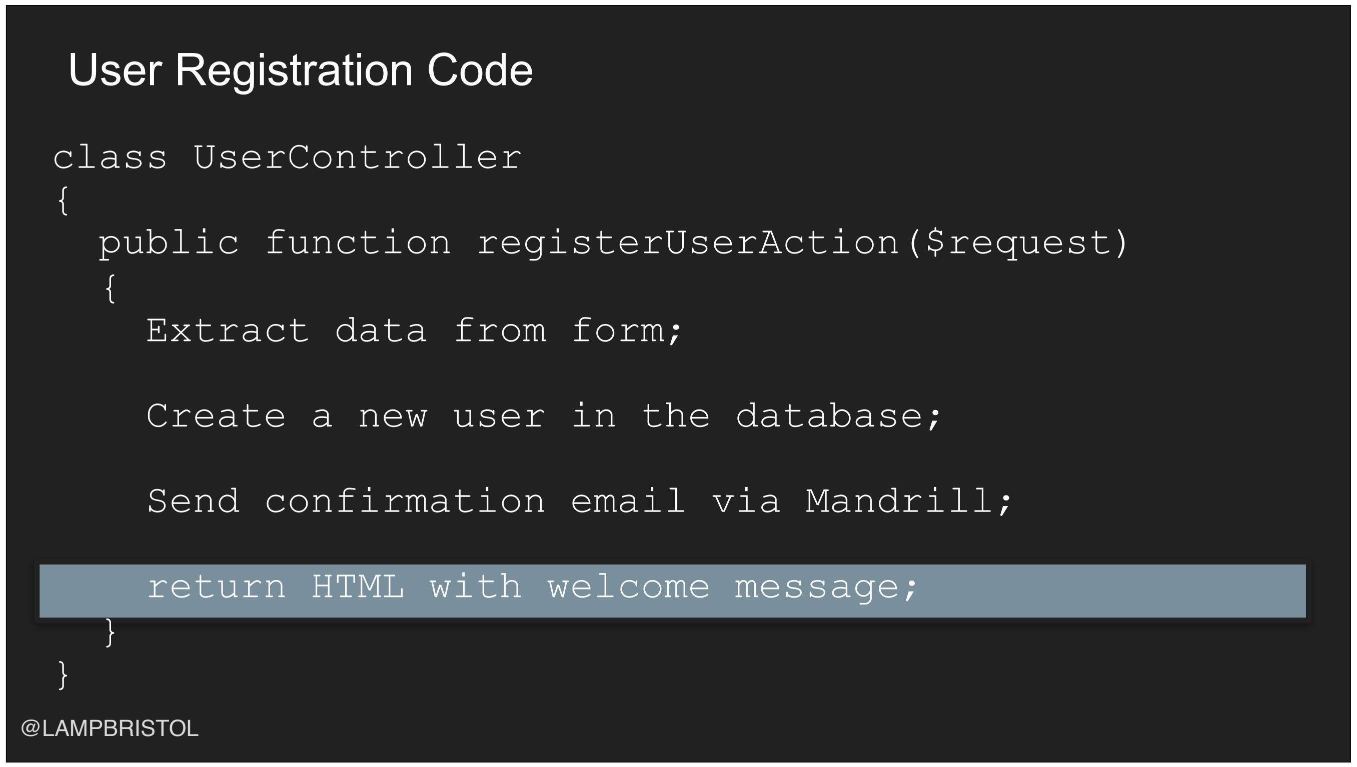
- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?

User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?

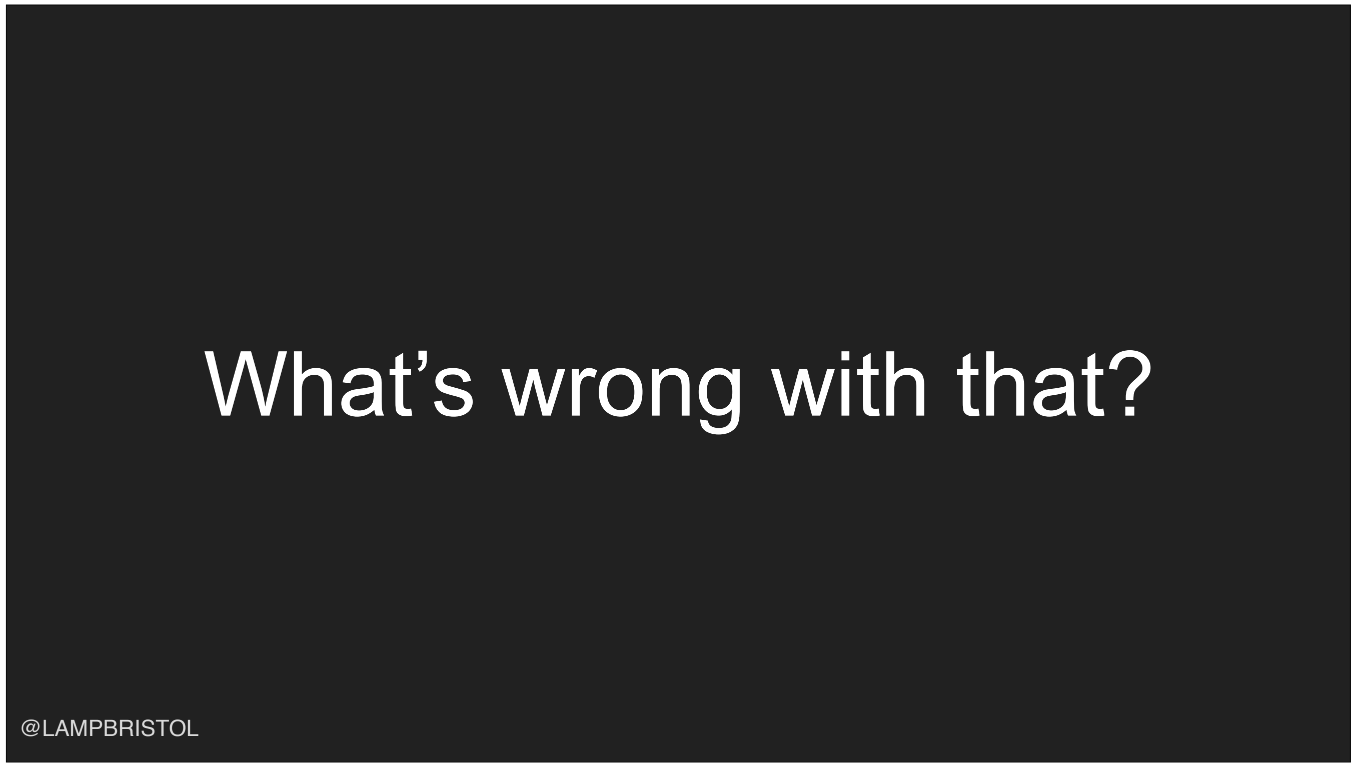


User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from form;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

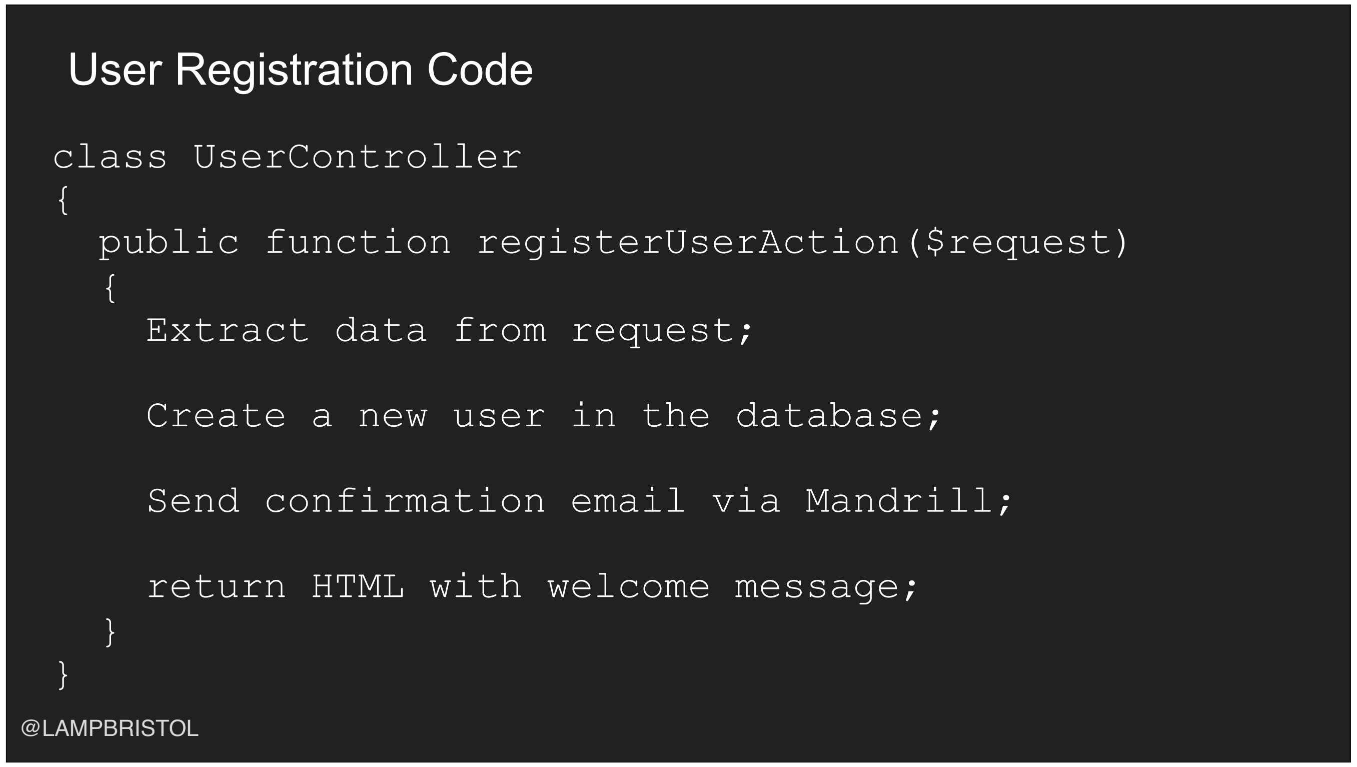
- Imagine we have some code that registers a new user.
- The controller is dealing with a website using HTML forms.
- How many have seen controllers like this?



What’s wrong with that?

@LAMPBRISTOL

Tightly coupled code.

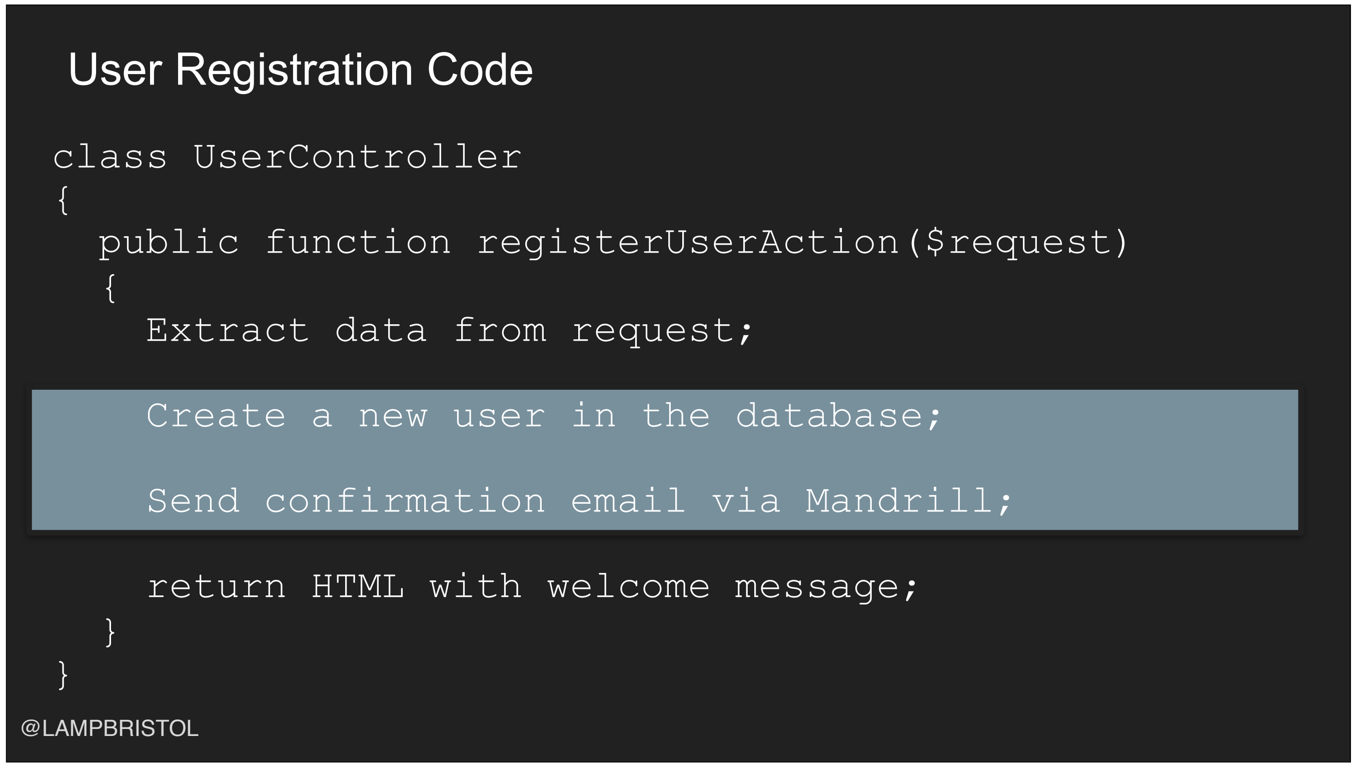


User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from request;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

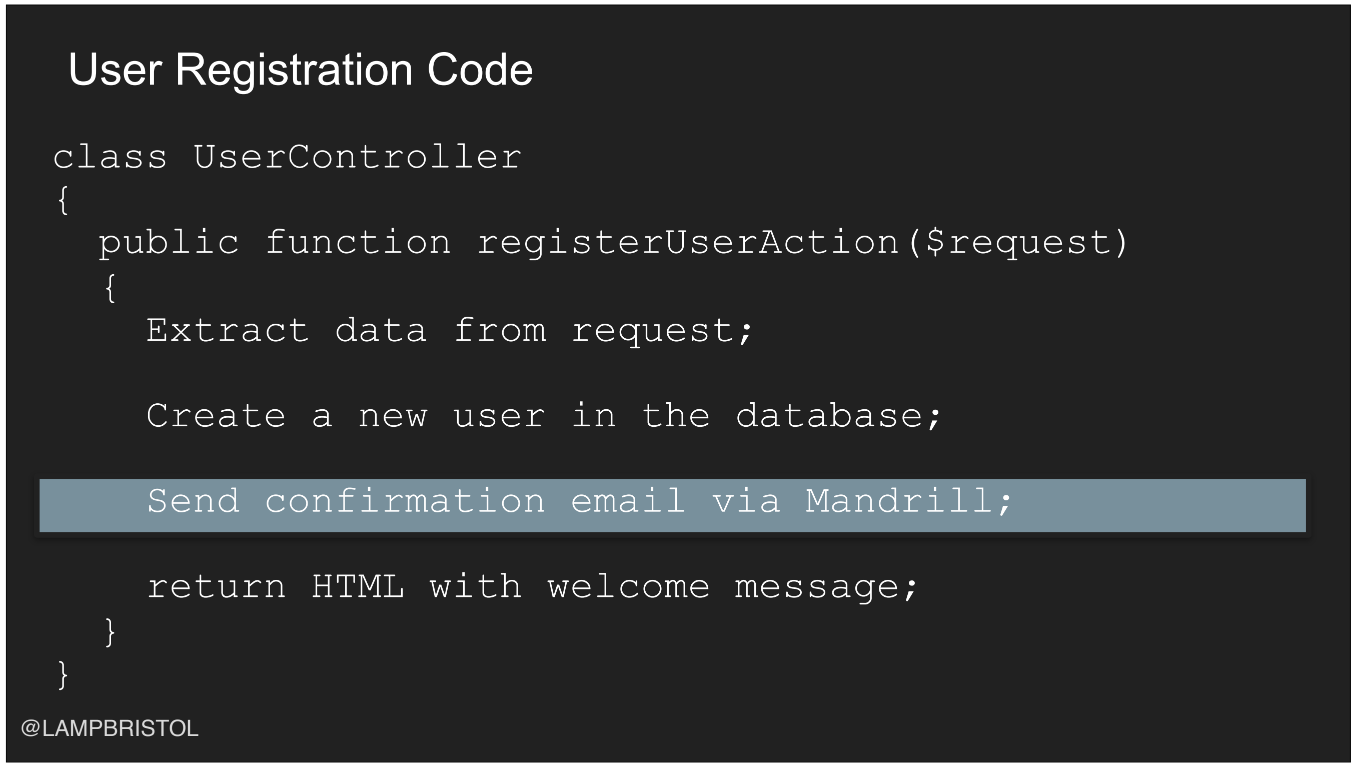
- What if we wanted a REST API (for a mobile app) as well as a web interface. It’s not easy. The common code we need to create a new user is in the middle of the controller code.
- What happens if we want to change how we send emails to move away from Mandrill (again difficult with the code on the screen).



User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from request;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

- What if we wanted a REST API (for a mobile app) as well as a web interface. It’s not easy. The common code we need to create a new user is in the middle of the controller code.
- What happens if we want to change how we send emails to move away from Mandrill (again difficult with the code on the screen).


User Registration Code

```
class UserController
{
  public function registerUserAction($request)
  {
    Extract data from request;

    Create a new user in the database;

    Send confirmation email via Mandrill;

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- What if we wanted a REST API (for a mobile app) as well as a web interface. It’s not easy. The common code we need to create a new user is in the middle of the controller code.
- What happens if we want to change how we send emails to move away from Mandrill (again difficult with the code on the screen).

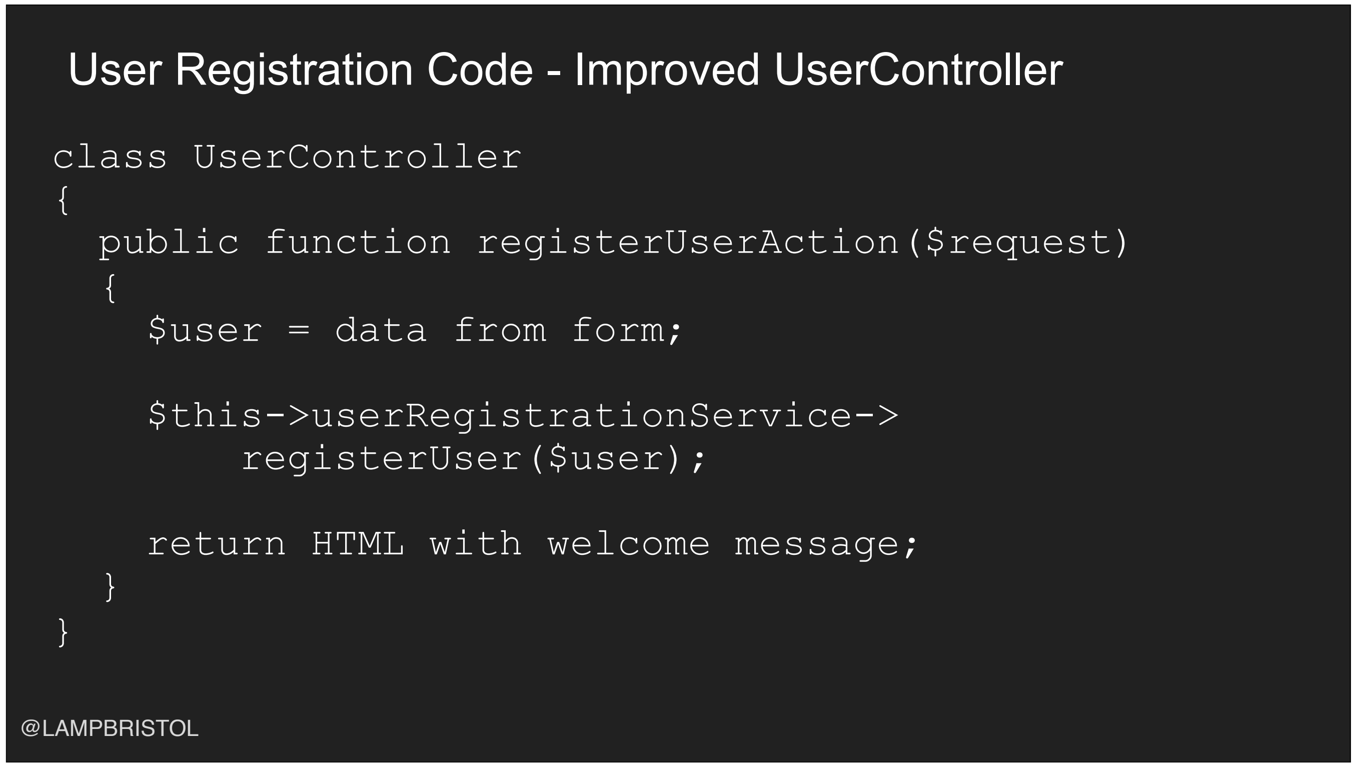
We want loosely couple code

User Registration Code - UserRegistrationService

```
interface UserRegistrationServiceInterface
{
  /**
   * Registers a new user
   *
   */
  public function registerUser(User $user);
}
```

@LAMPBRISTOL

- Remove user registration logic from controller.
- Move it to a class that implements this interface. All controller cares about is registering a new user not how the new user is registered.
- This service can be reused by a controller for a REST endpoint , or even importing users via a CLI.

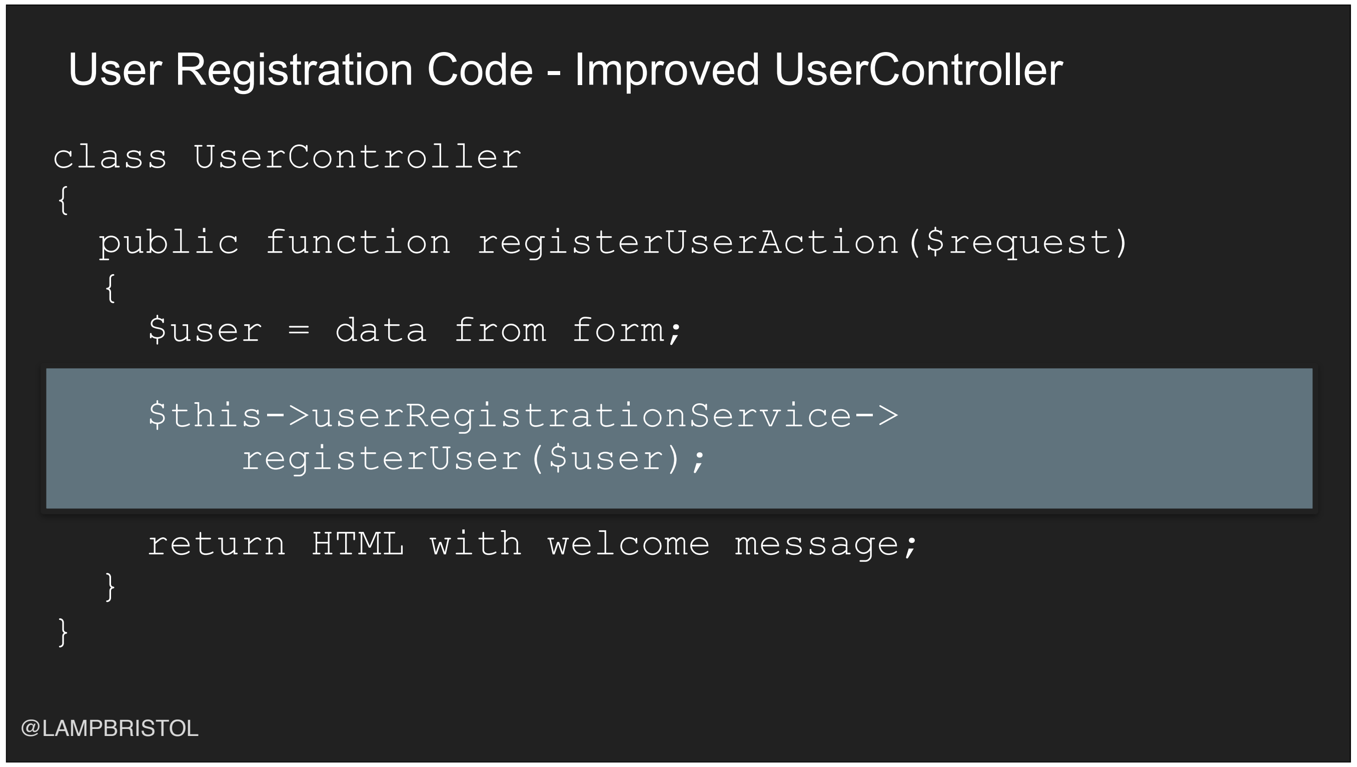


User Registration Code - Improved UserController

```
class UserController
{
  public function registerUserAction($request)
  {
    $user = data from form;

    $this->userRegistrationService->
        registerUser($user);

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- Controller now takes advantage of user registration service.

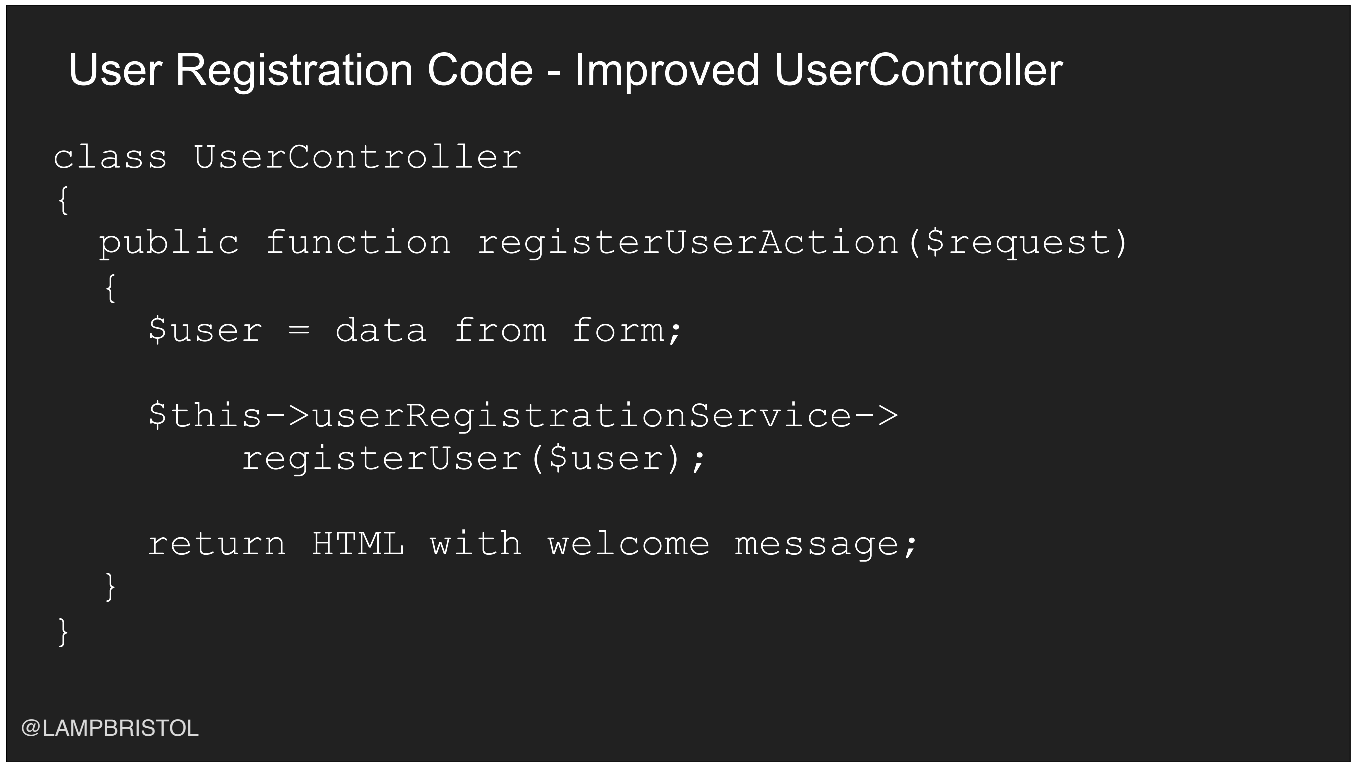


User Registration Code - Improved UserController

```
class UserController
{
  public function registerUserAction($request)
  {
    $user = data from form;

    $this->userRegistrationService->
        registerUser($user);

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- Controller now takes advantage of user registration service.

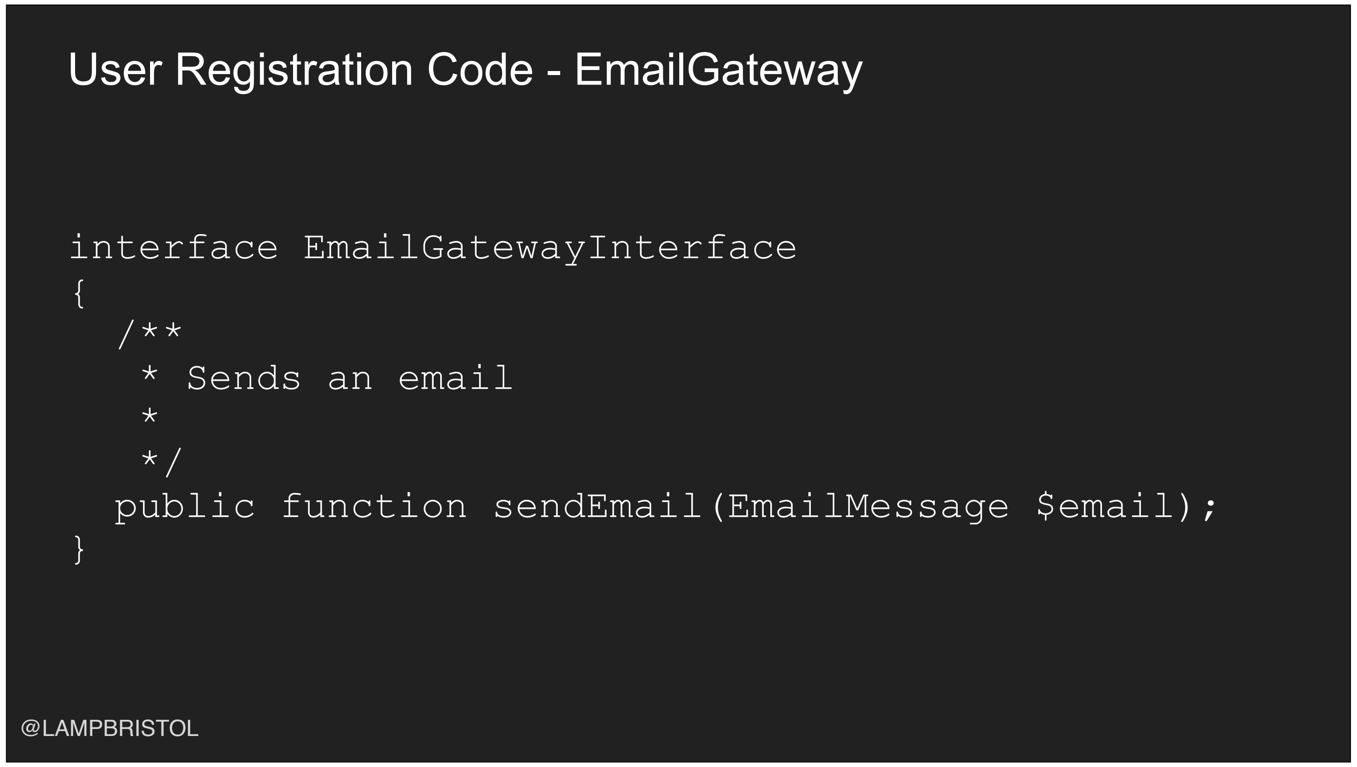


User Registration Code - Improved UserController

```
class UserController
{
  public function registerUserAction($request)
  {
    $user = data from form;

    $this->userRegistrationService->
        registerUser($user);

    return HTML with welcome message;
  }
}
```

@LAMPBRISTOL

- Controller now takes advantage of user registration service.

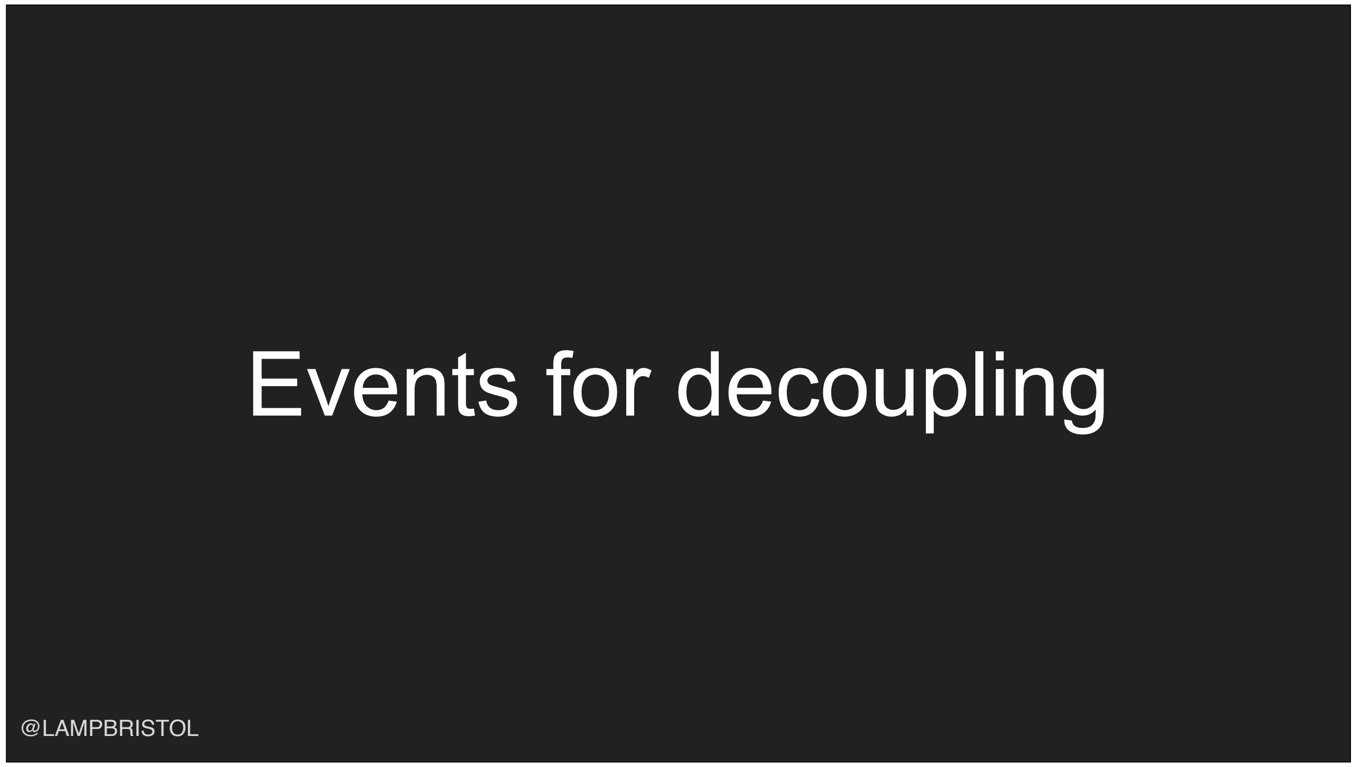


User Registration Code - EmailGateway

```
interface EmailGatewayInterface
{
  /**
   * Sends an email
   *
   */
  public function sendEmail(EmailMessage $email);
}
```

@LAMPBRISTOL

- User registration code makes use of sending emails. All it cares about is that an email is sent, not how, or which email gateway is used. This interface hides that detail.
-Code is much more decoupled



Events for decoupling

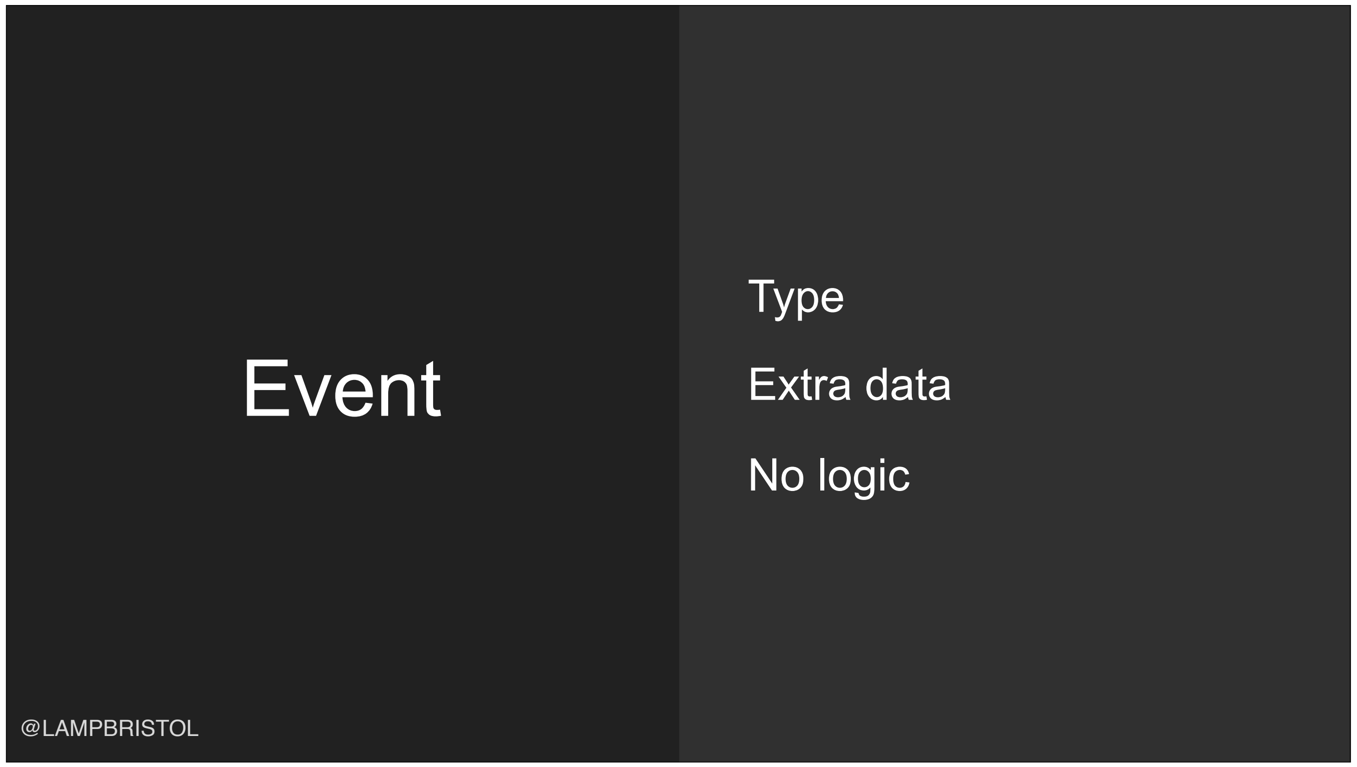
@LAMPBRISTOL

Events can be used for decoupling.
It is not appropriate in all cases but it’s a tool you should have in your tool box.

What is an Event?

@LAMPBRISTOL

Event is an action recognised by software that may be handled by software.
Examples include:
- Mouse click
- New User Registration
- Order placed
-



Event

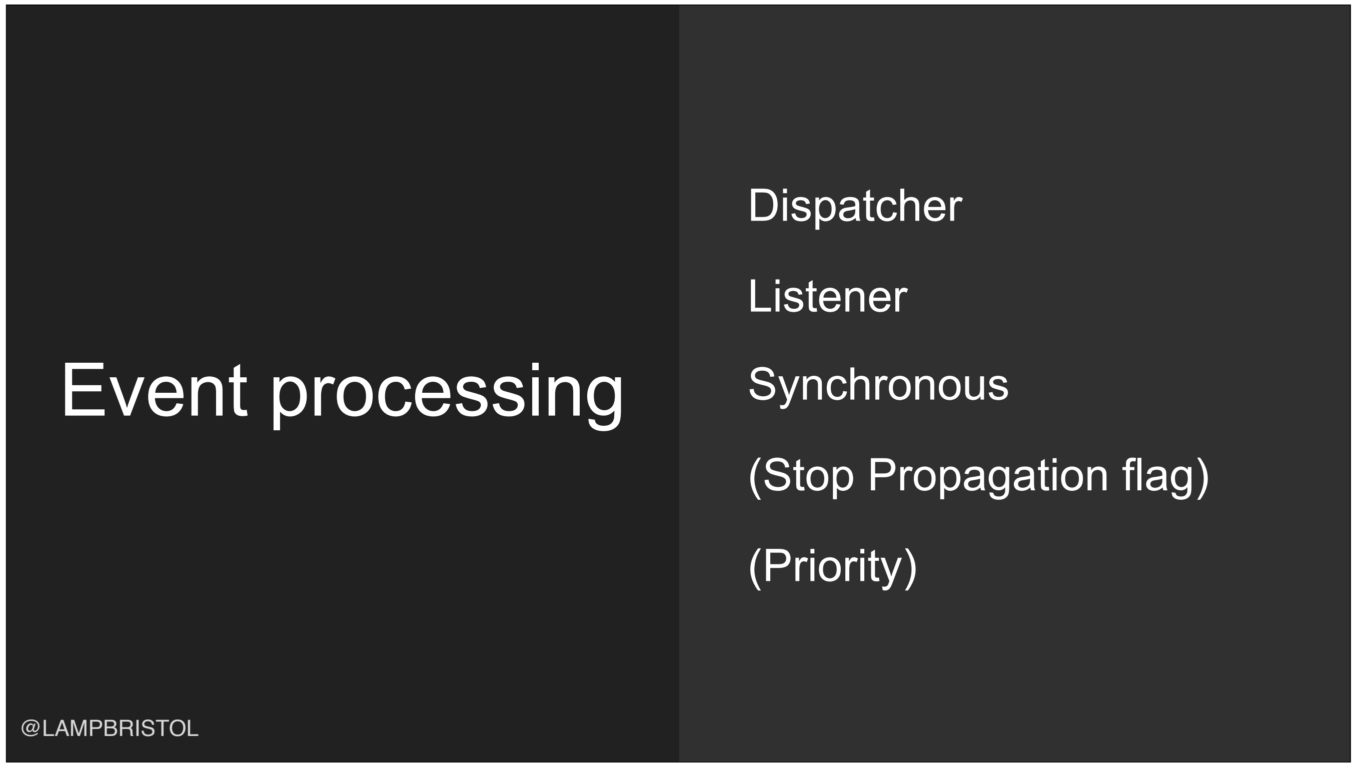
Type

Extra data

No logic

@LAMPBRISTOL

Event
Type: What does this event represent (e.g. NewUserRegistrationEvent)
Extra Data: Information passed by the event source to the event handler
The object should only hold data, there should be no logic in it.



Event processing

Dispatcher

Listener

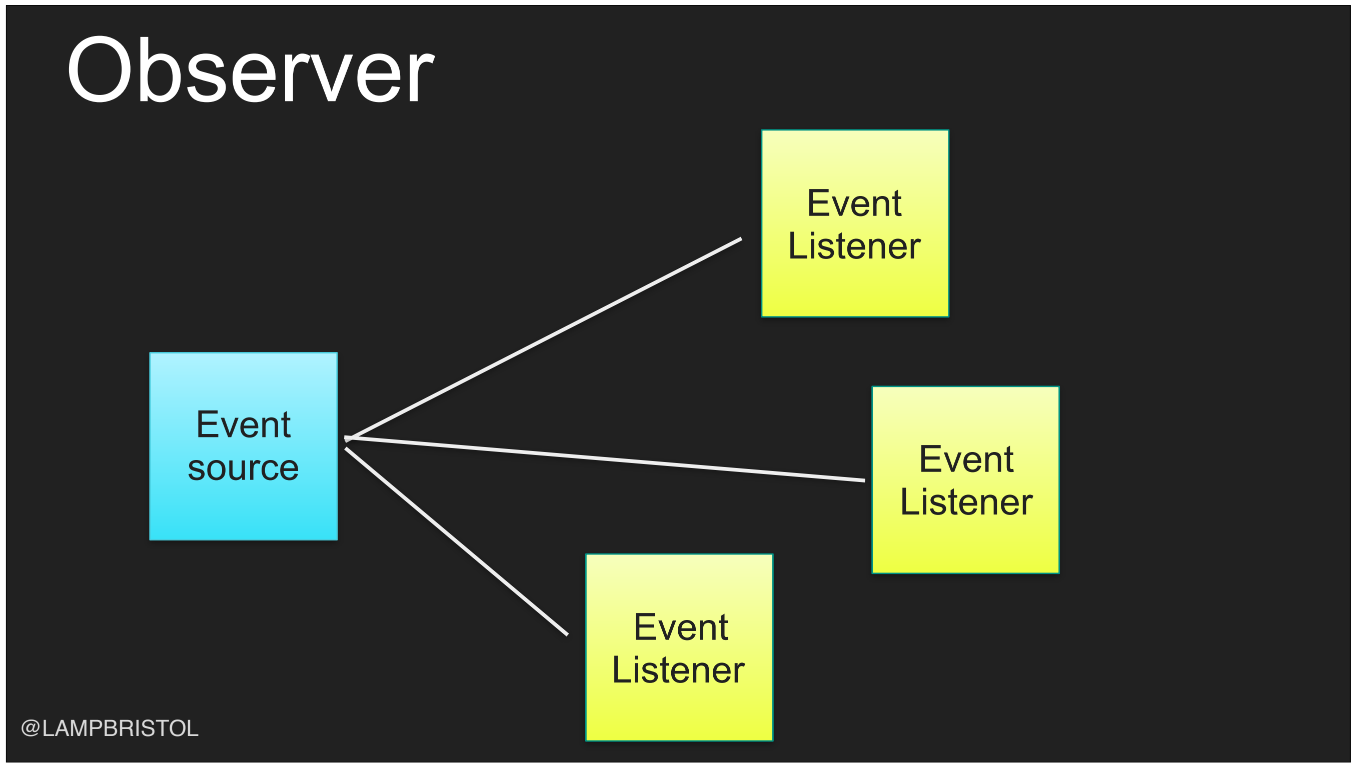
Synchronous

(Stop Propagation flag)

(Priority)

@LAMPBRISTOL

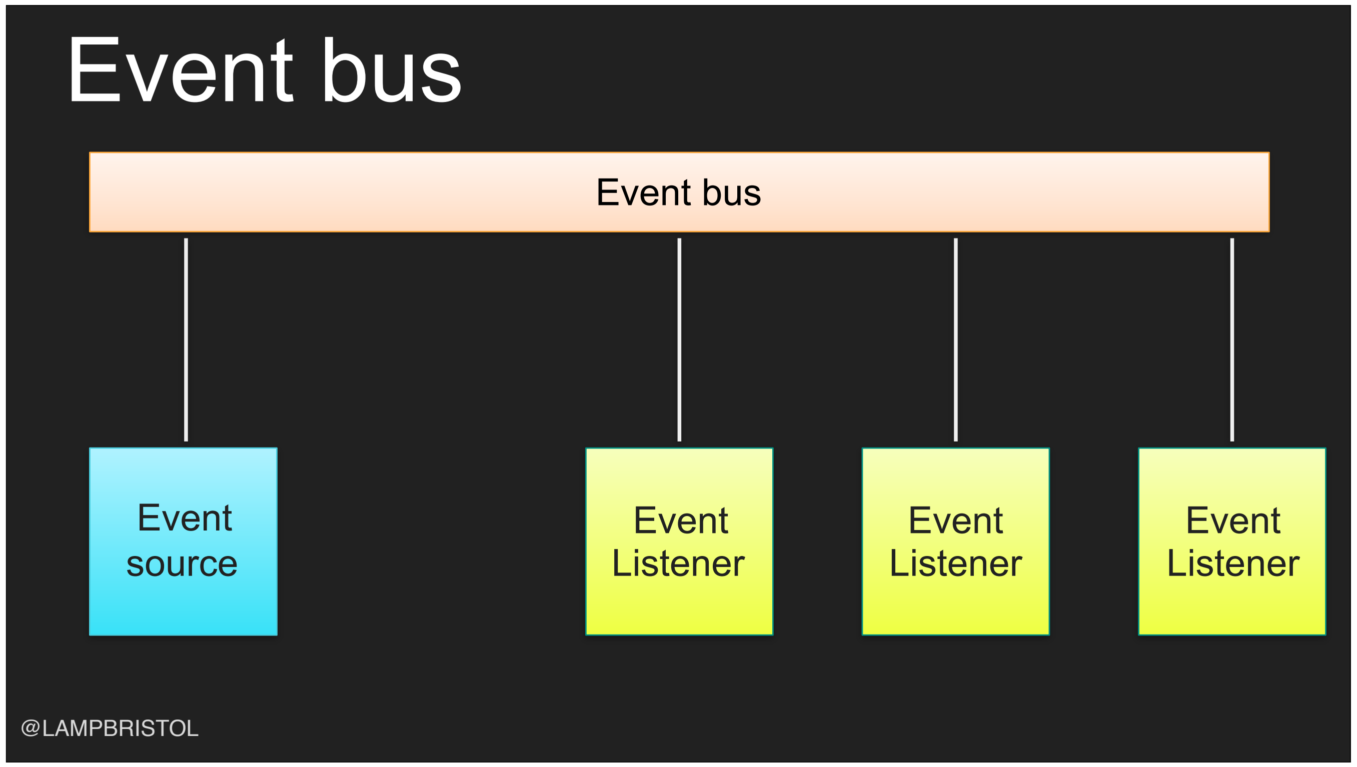
- Events are ‘created’ and sent to the event processing framework via a ‘dispatcher’
- Code that responds to events are ‘Listeners’
- For the purpose of examples used in this talk events are processed synchronously
- Following are optional - not all event systems support these:
- Events might have a stop propagation flag. Once set and subsequent listeners are not called. Instead flow of code returns to the line after event was dispatched.
- Event listeners may have a priority. High priority event listeners are run first.



Observer

@LAMPBRISTOL

- Event listeners register themselves with the event source (or dispatcher)
- Listeners much know about the source
- Event source doesn't really care who is listening or have any knowledge of what the listeners do
- Some level of decoupling (which might be enough for the problem you're trying to solve)
- E.g. listening to click events on a DOM element in JavaScript



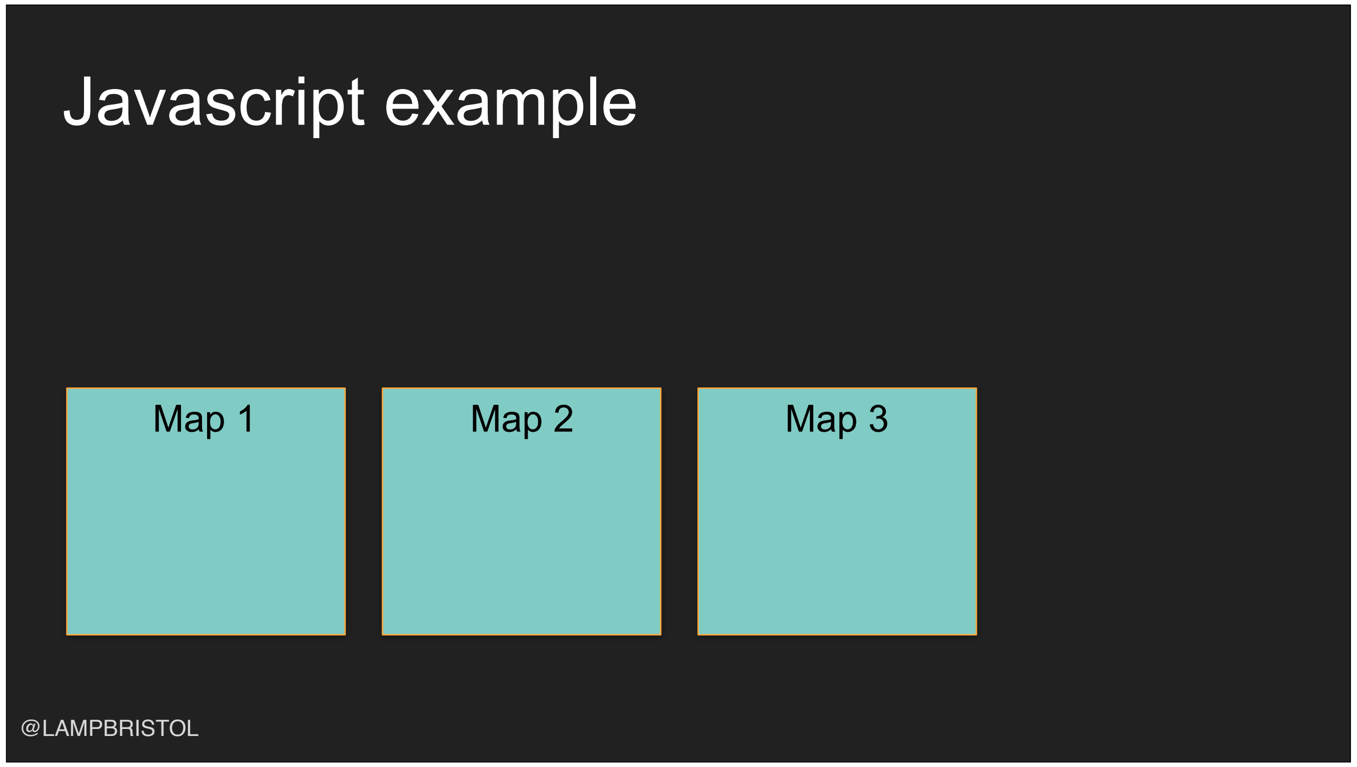
Event bus

- Event bus mediates between dispatchers and listeners.
- Dispatcher pushes messages to the event bus
- Listeners register their interest in events of a certain type.
- Event bus passes events from the dispatcher to the interested listeners
- Fully decoupled
- Dispatcher doesn't know who, if anyone, is listening
- Listeners done't know who is dispatching events.

Events in action

@LAMPBRISTOL

We will look at a few examples. The first being a Javascript one and the 2nd (and 3rd if there is time) PHP ones.

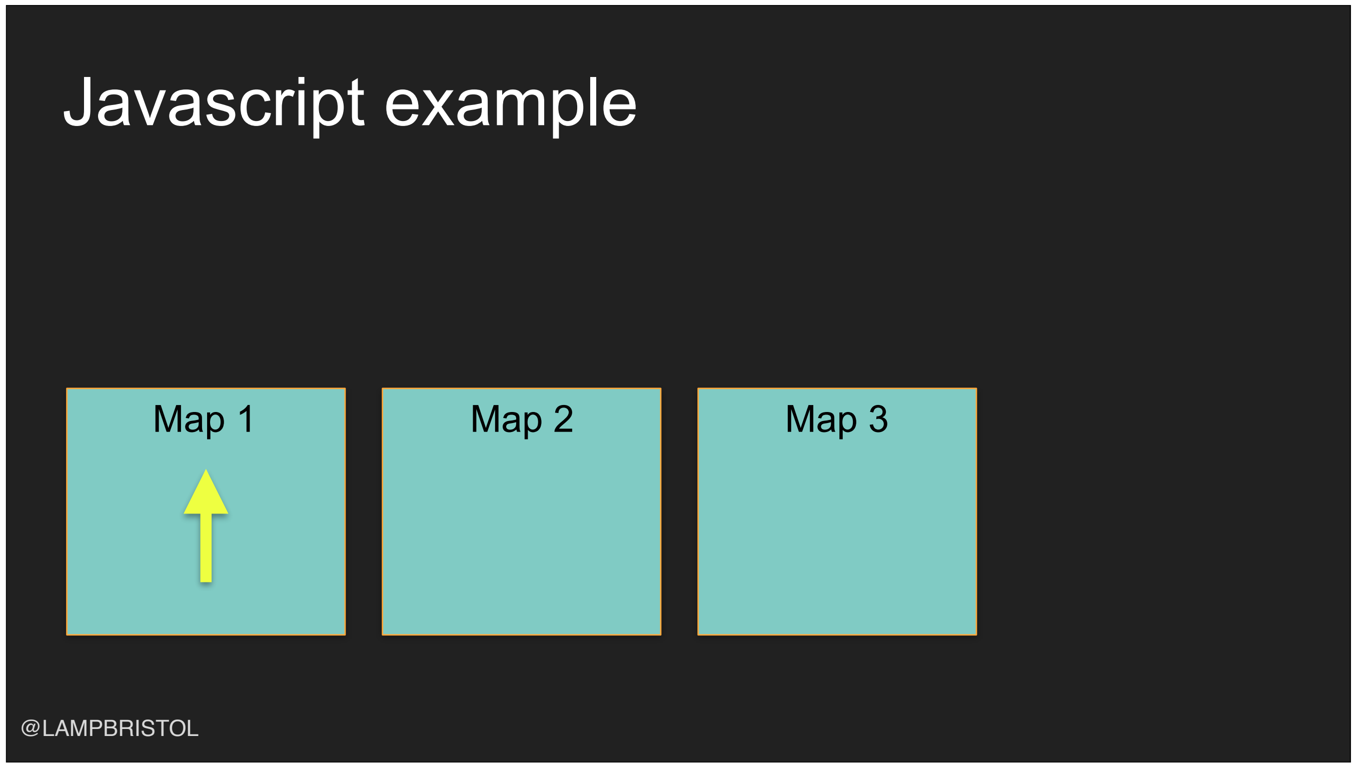


Javascript example

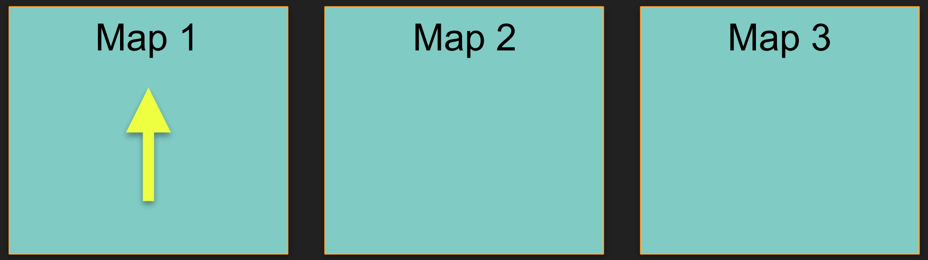
Map 1
Map 2
Map 3

@LAMPBRISTOL

- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.

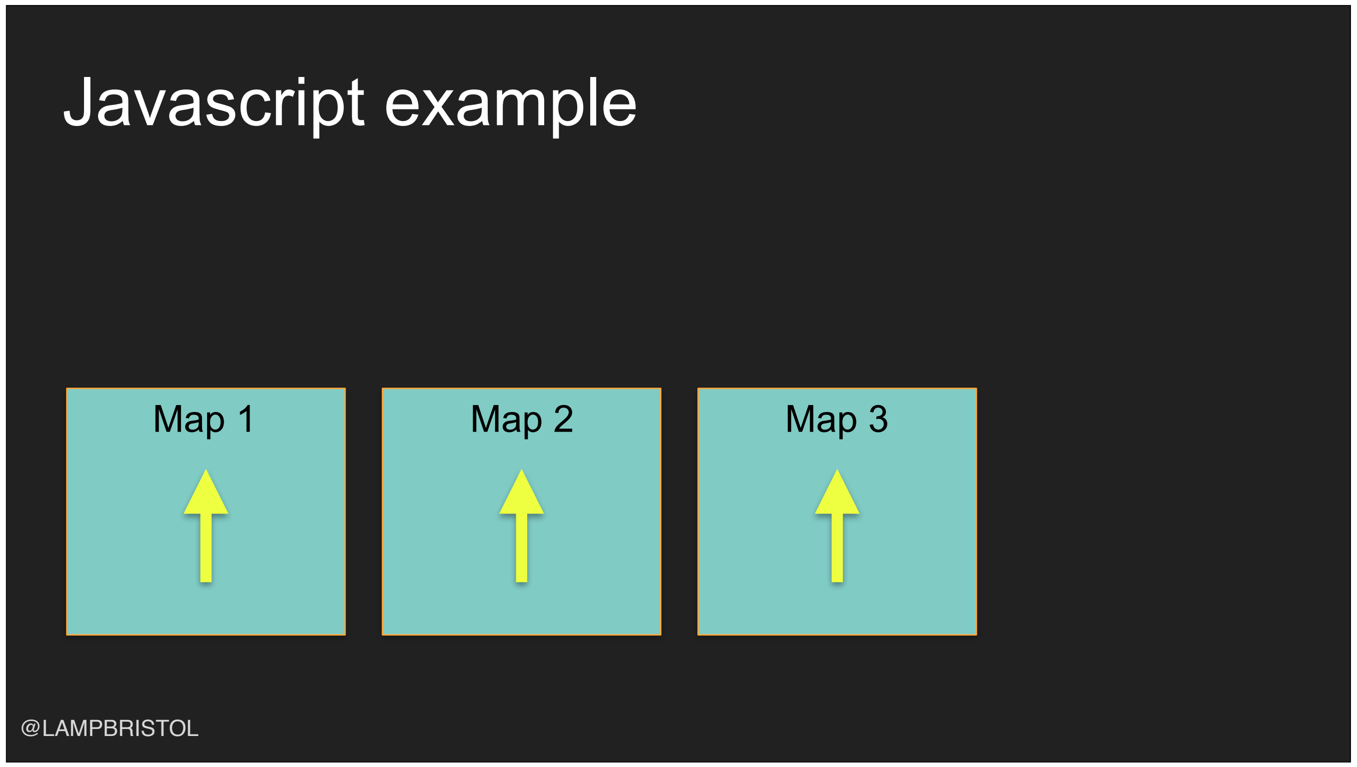


Javascript example



@LAMPBRISTOL

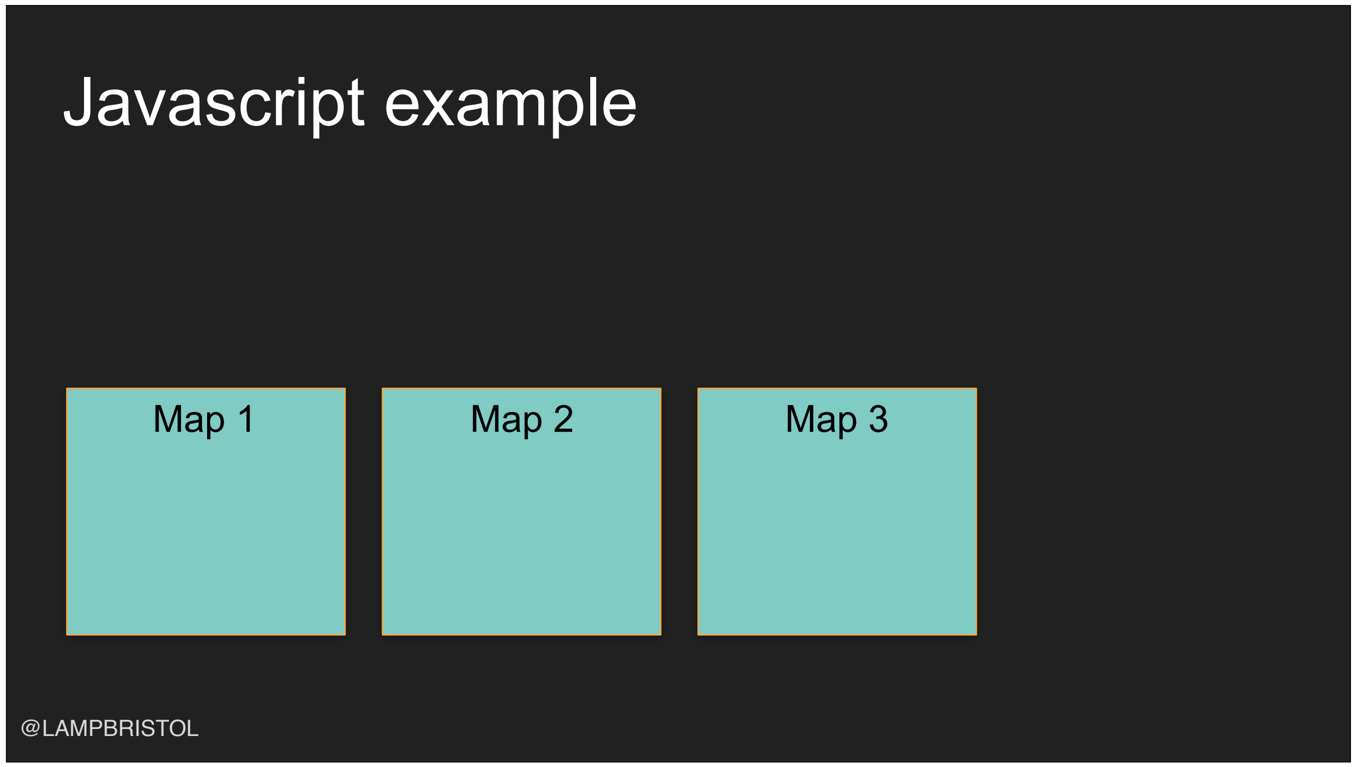
- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.



Javascript example

@LAMPBRISTOL

- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.



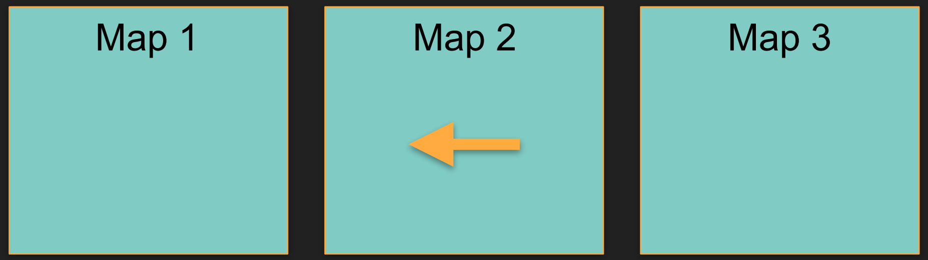
Javascript example

Map 1
Map 2
Map 3

@LAMPBRISTOL

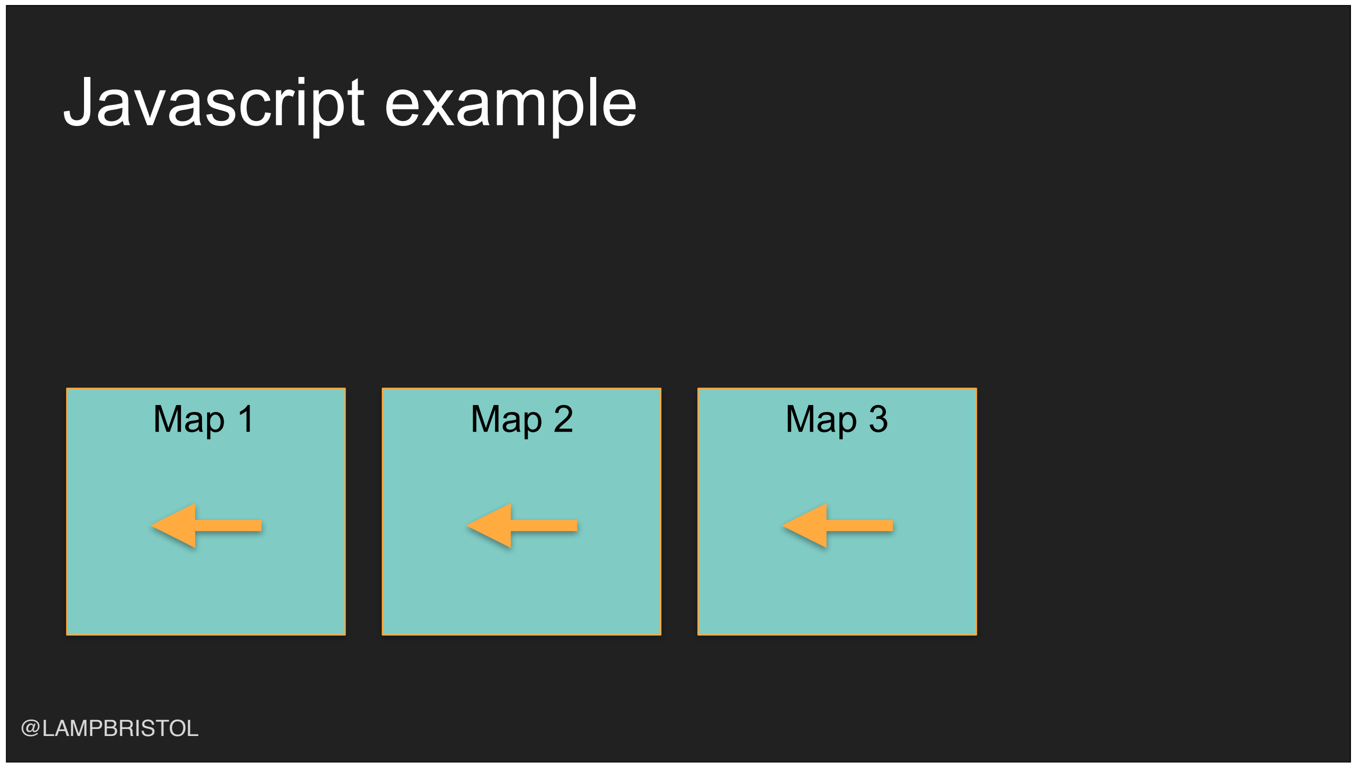
- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.

Javascript example



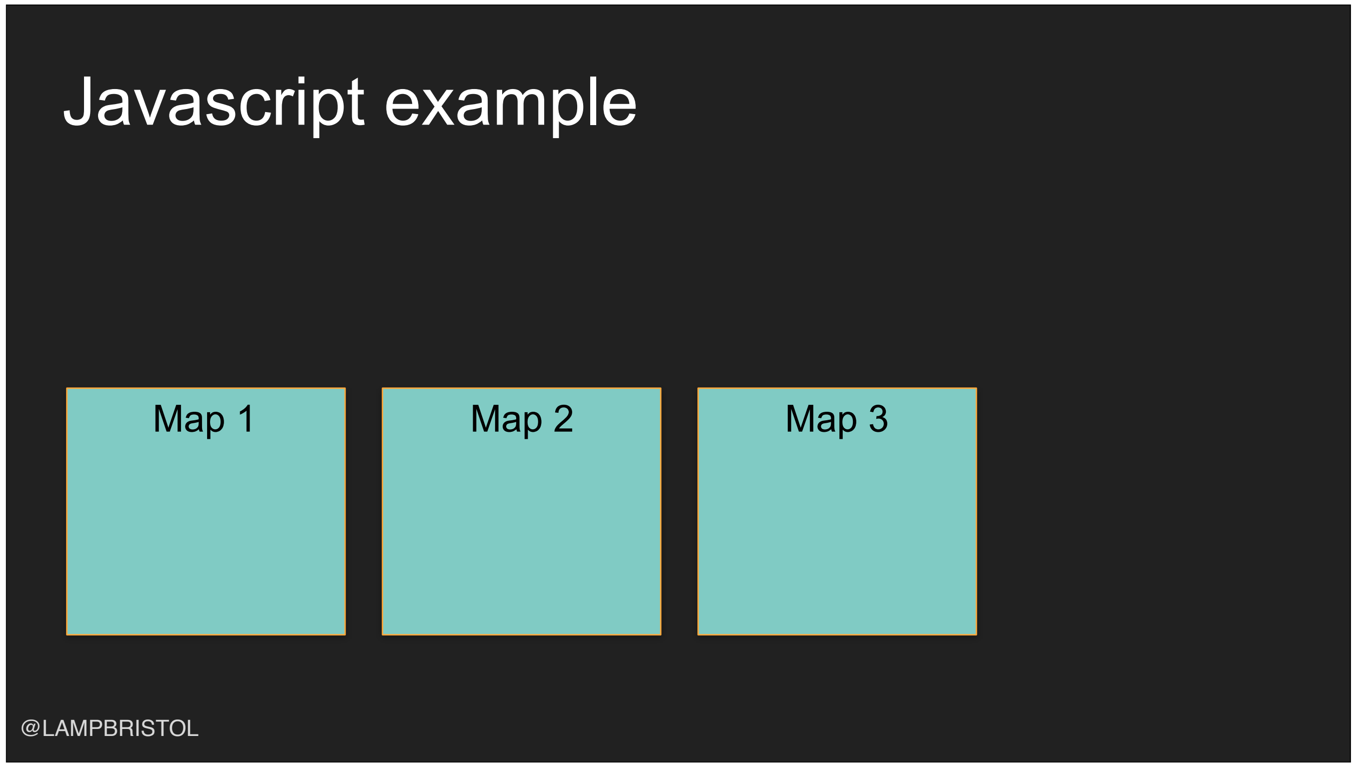
@LAMPBRISTOL

- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.



Javascript example

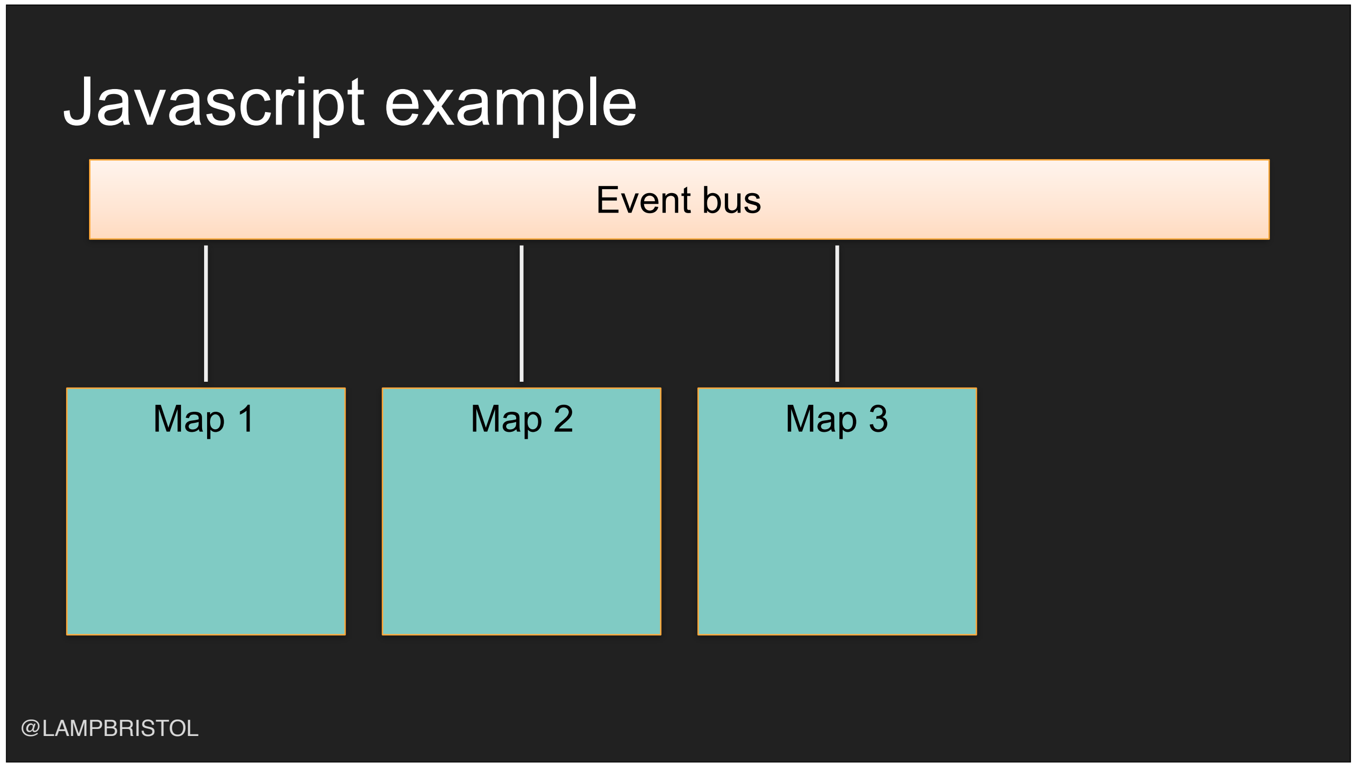
- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.



Javascript example

@LAMPBRISTOL

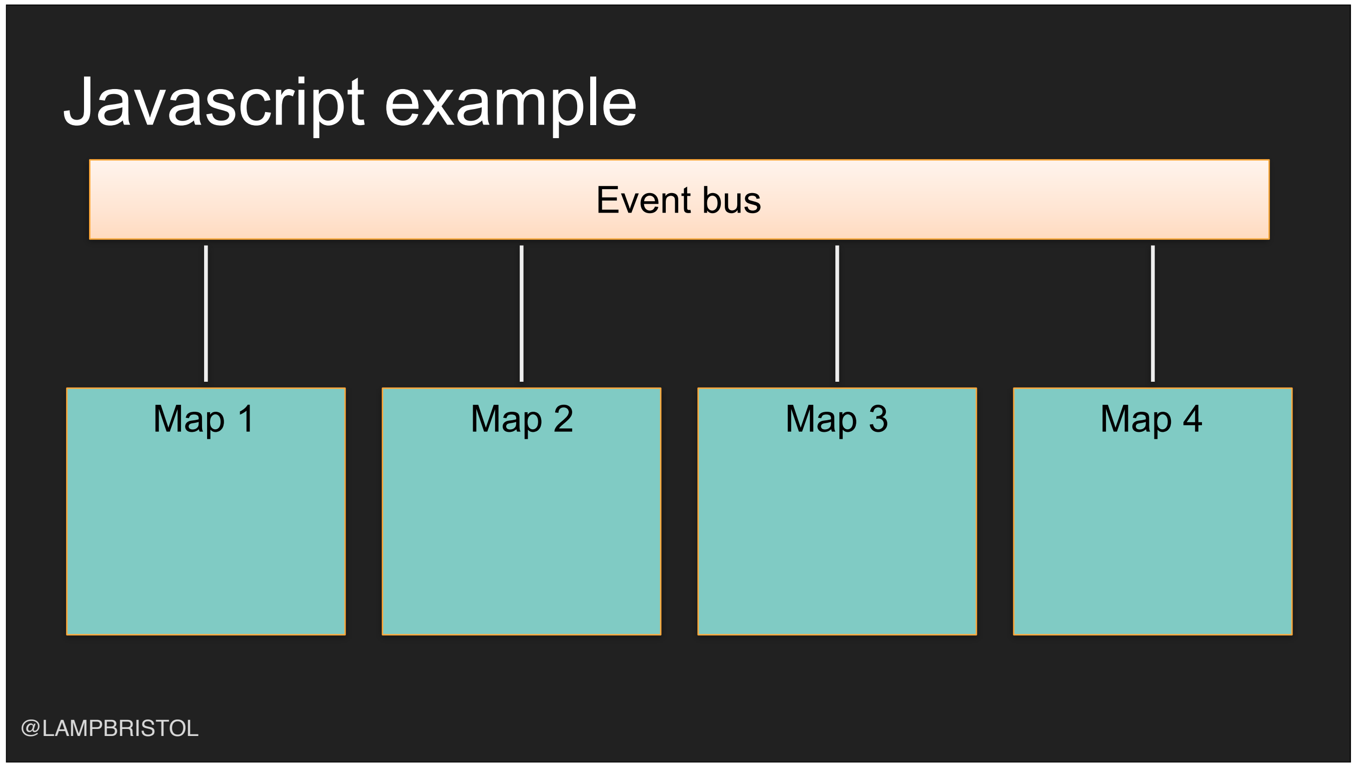
- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.



Javascript example

@LAMPBRISTOL

- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.



Javascript example

@LAMPBRISTOL

- Each map showed a different data layer (e.g. perception of crime, actual crime, etc)
- Each map had to show the same geographic area (so comparisons could be made).
- A user could interact with any map.
- Moving map 1 to the left meant the other maps must also move to the left.
- Moving map 2 up meant the other maps must also move up.
- Each map registered itself with the event bus.
- Each map could dispatch and listen for MoveMapEvent
- One map would move, all the others would update.
- This could scale to any number of maps, they just needed registering with the event bus.

User Registration Example

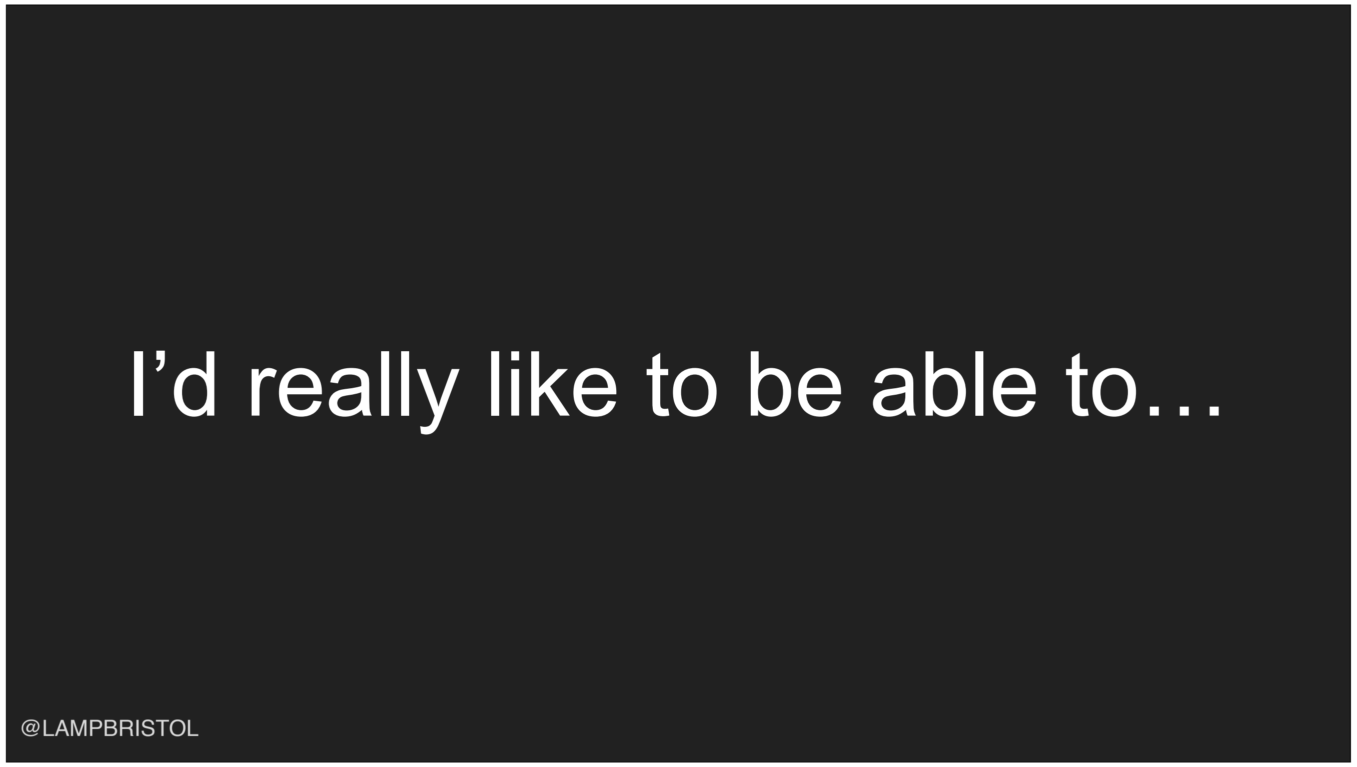
@LAMPBRISTOL

Remember our user registration example from the start. This is common code so we created a user library registration library.

User Registration Library

@LAMPBRISTOL

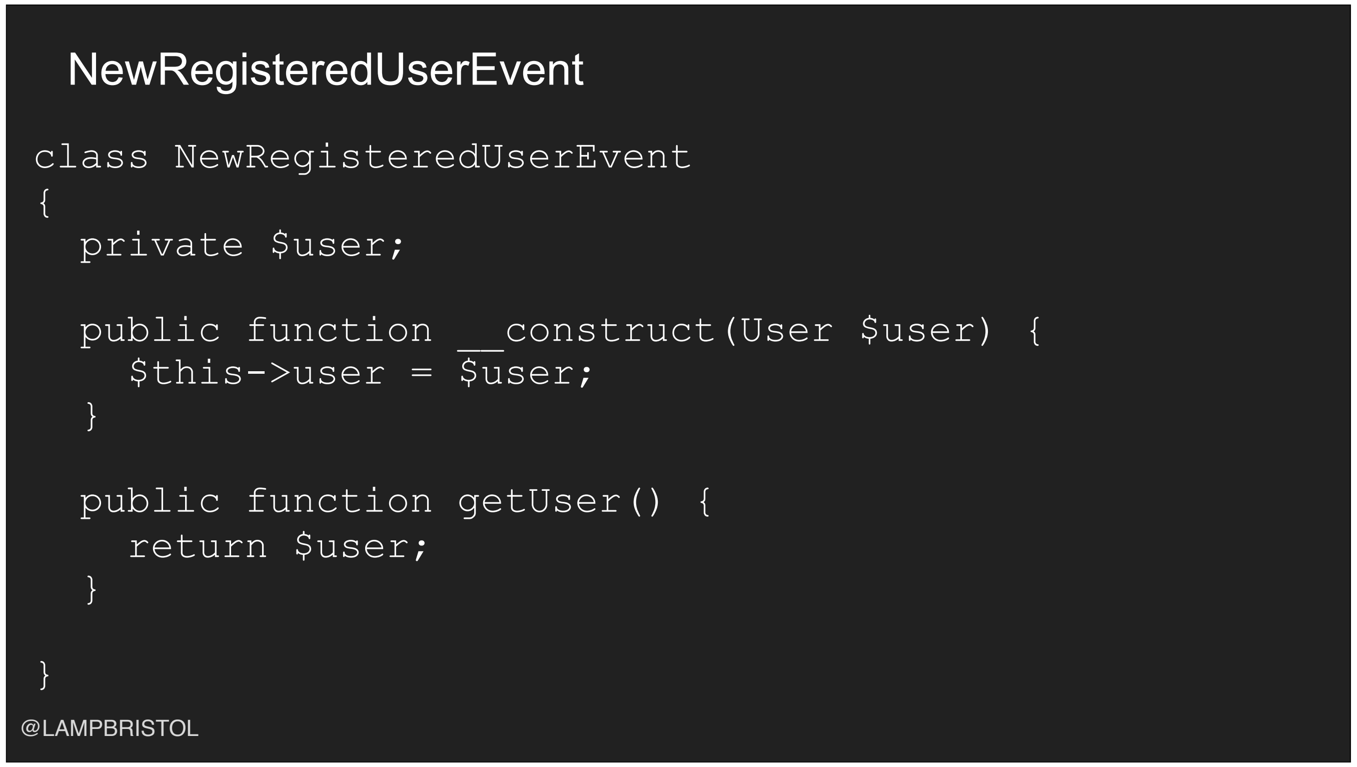
It was used on lots of projects. All the customers are happy..



I’d really like to be able to…

@LAMPBRISTOL

- Then one day a customer wants to be able to do something extra after a user has completed registration. This functionality is specific only to this one customer. Let’s say they wanted to send newly registered users a personalised discount voucher.
- How do we do that without having lots of conditional code in our library?

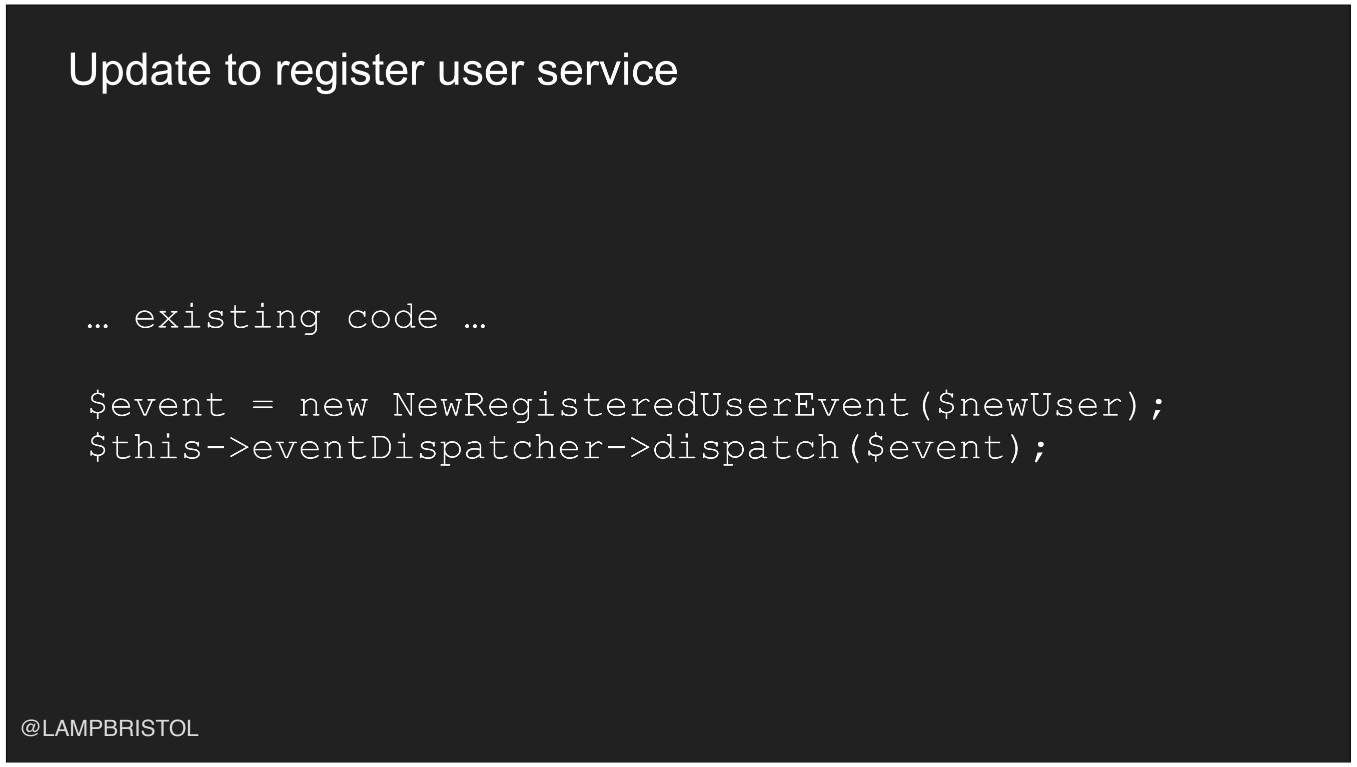


NewRegisteredUserEvent

```
class NewRegisteredUserEvent
{
  private $user;

  public function __construct(User $user) {
    $this->user = $user;
  }

  public function getUser() {
    return $user;
  }

}
```

@LAMPBRISTOL

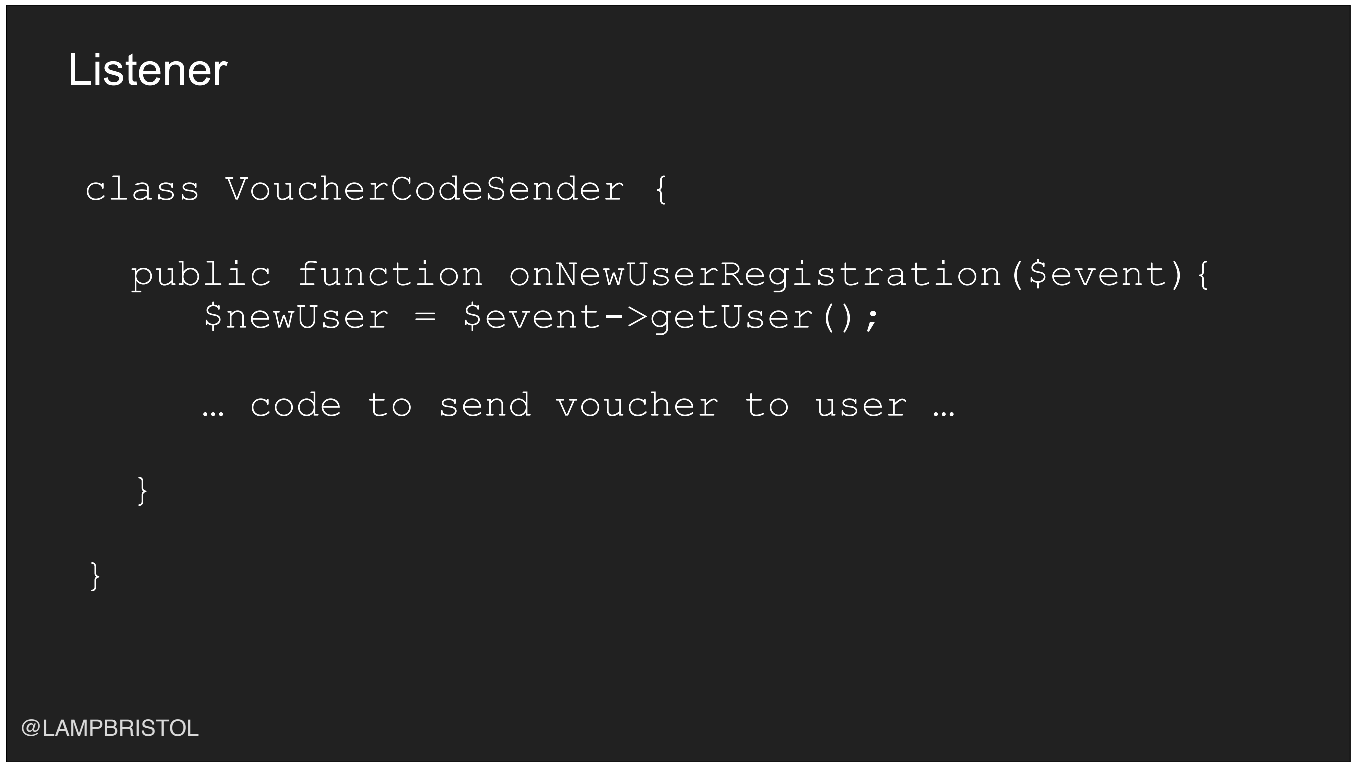
We could create a event that gets dispatched after a new user has been registered. It would hold details about the newly registered user


Update to register user service

```
… existing code …

$event = new NewRegisteredUserEvent($newUser);
$this->eventDispatcher->dispatch($event);
```

@LAMPBRISTOL

The register user service is updated to dispatch an this event after the new user has been created.

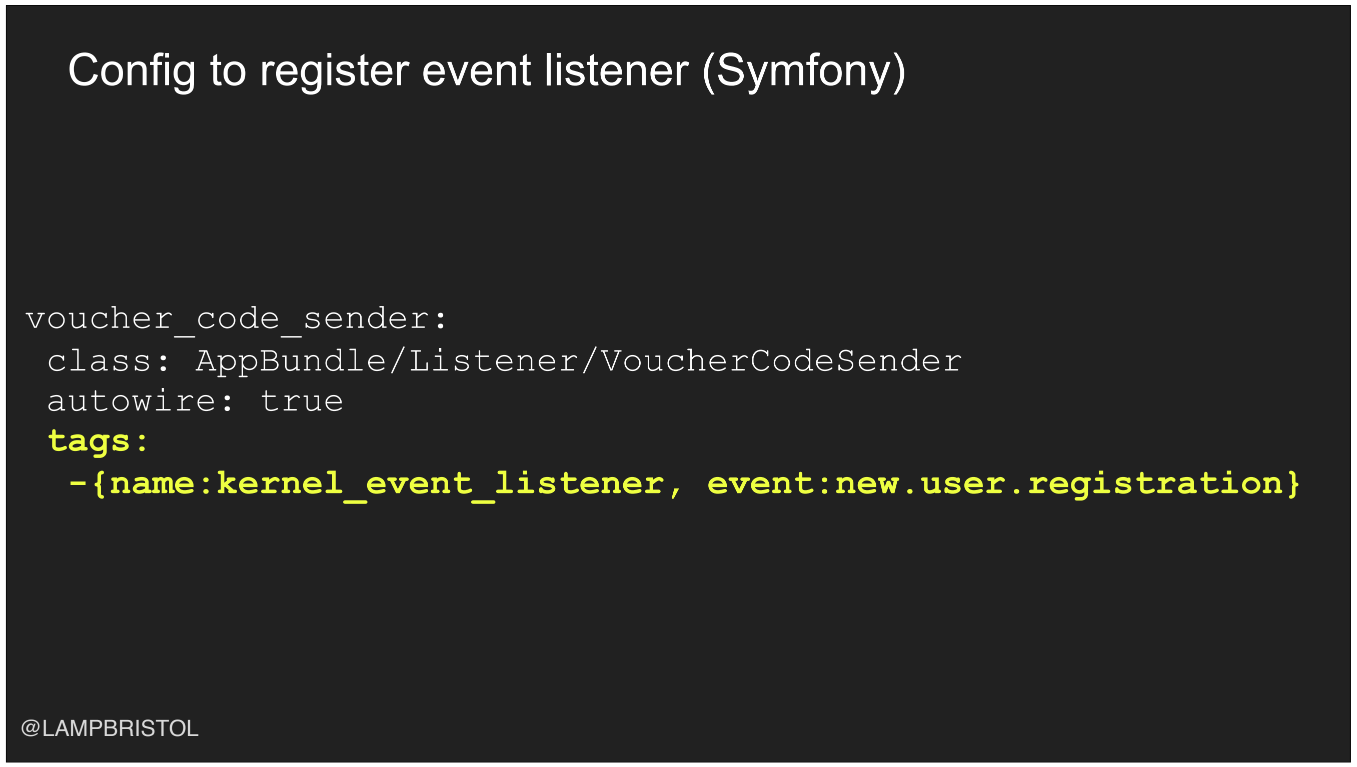


Listener

```
class VoucherCodeSender {

  public function onNewUserRegistration($event){
     $newUser = $event->getUser();

     … code to send voucher to user …

  }

}
```

@LAMPBRISTOL

Code can listen for these events and hook in extra functionality at these points. E.g. VoucherCodeSender



Config to register event listener (Symfony)

```
voucher_code_sender:
 class: AppBundle/Listener/VoucherCodeSender
 autowire: true
 tags:
  -{name:kernel_event_listener, event:new.user.registration}
```

@LAMPBRISTOL

Frameworks might provide hooks to achieve this easily. The code above is from Symfony configuration. Only the code in yellow is additional code to wire up the event listener on the previous page.

The ‘O’ in SOLID

@LAMPBRISTOL

We’ve used events to make our code open to extension but closed to modification. This is the Open/Closed principle from solid.
Other customers can come along with similar requirements, all the need to do is hook to to the NewRegisteredUserEvent.

Pros

Decoupled code

Good for extension

Cons

Less obvious

Harder to debug

Maybe no one is listening

Summary

Decoupling is good

Events can help

Good for extension points

No silver bullet

Only just the start

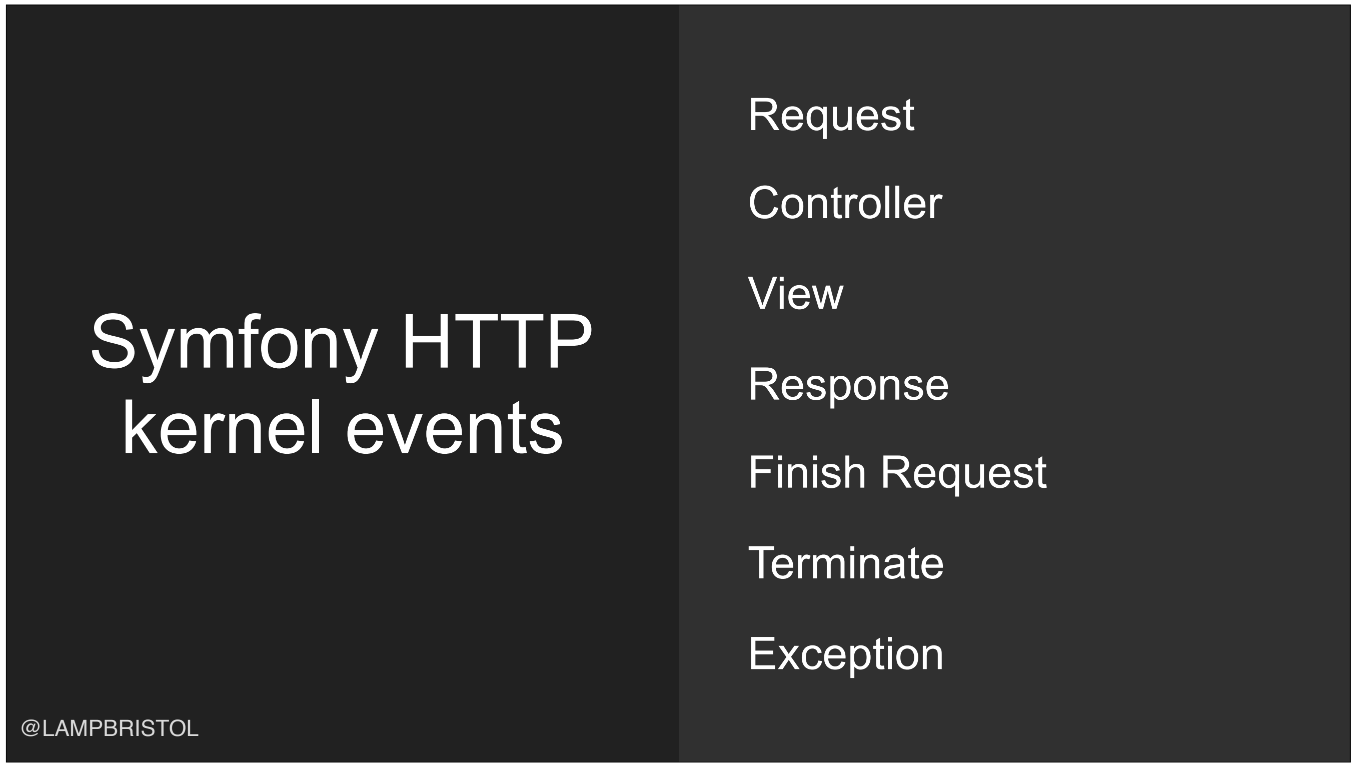
Questions

https://github.com/symfony/event-dispatcher

@LAMPBRISTOL

Bonus material

@LAMPBRISTOL



Symfony HTTP kernel events

Request

Controller

View

Response

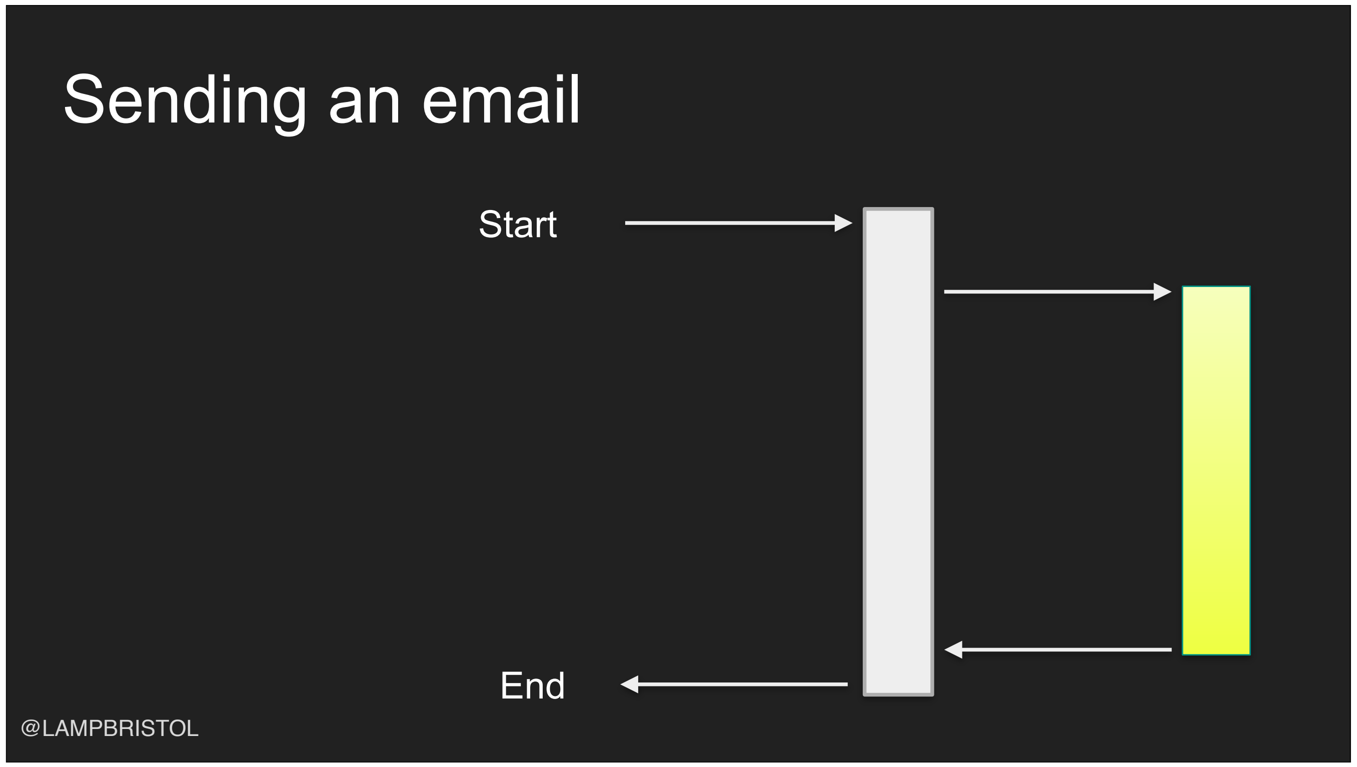
Finish Request

Terminate

Exception

@LAMPBRISTOL

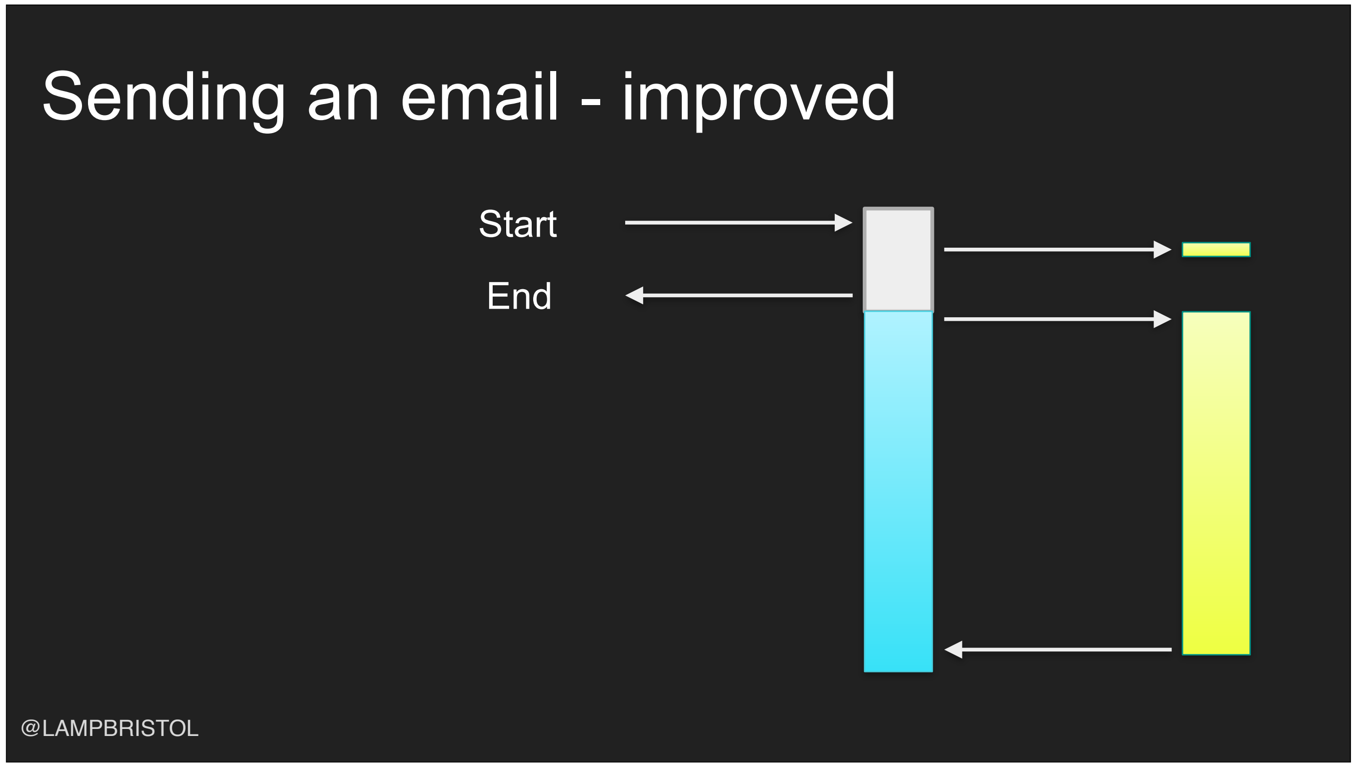
These are events that get dispatched by the HTTP kernel when it is processing an HTTP request. You can attach your own listeners to modify behaviour at each step.



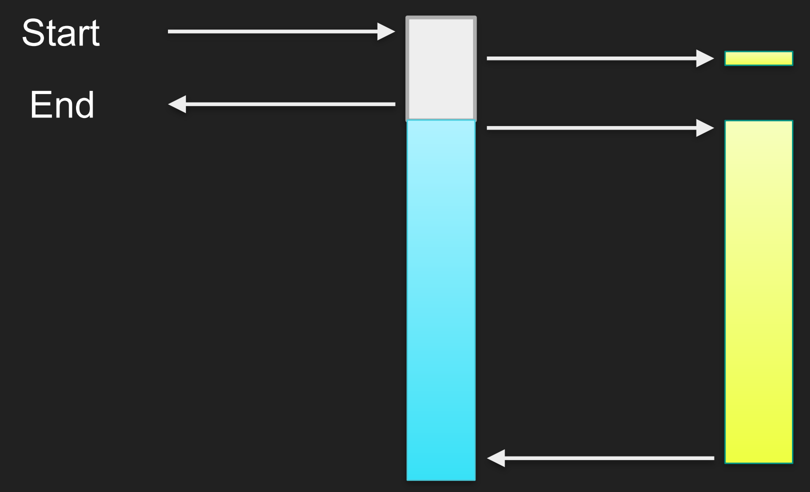
Sending an email

@LAMPBRISTOL

Imagine the user registration process. We create a new recored in the database (quick).
Then send an email, which will probably mean a network request (slow).
User has to wait around for a long time.



Sending an email - improved



@LAMPBRISTOL

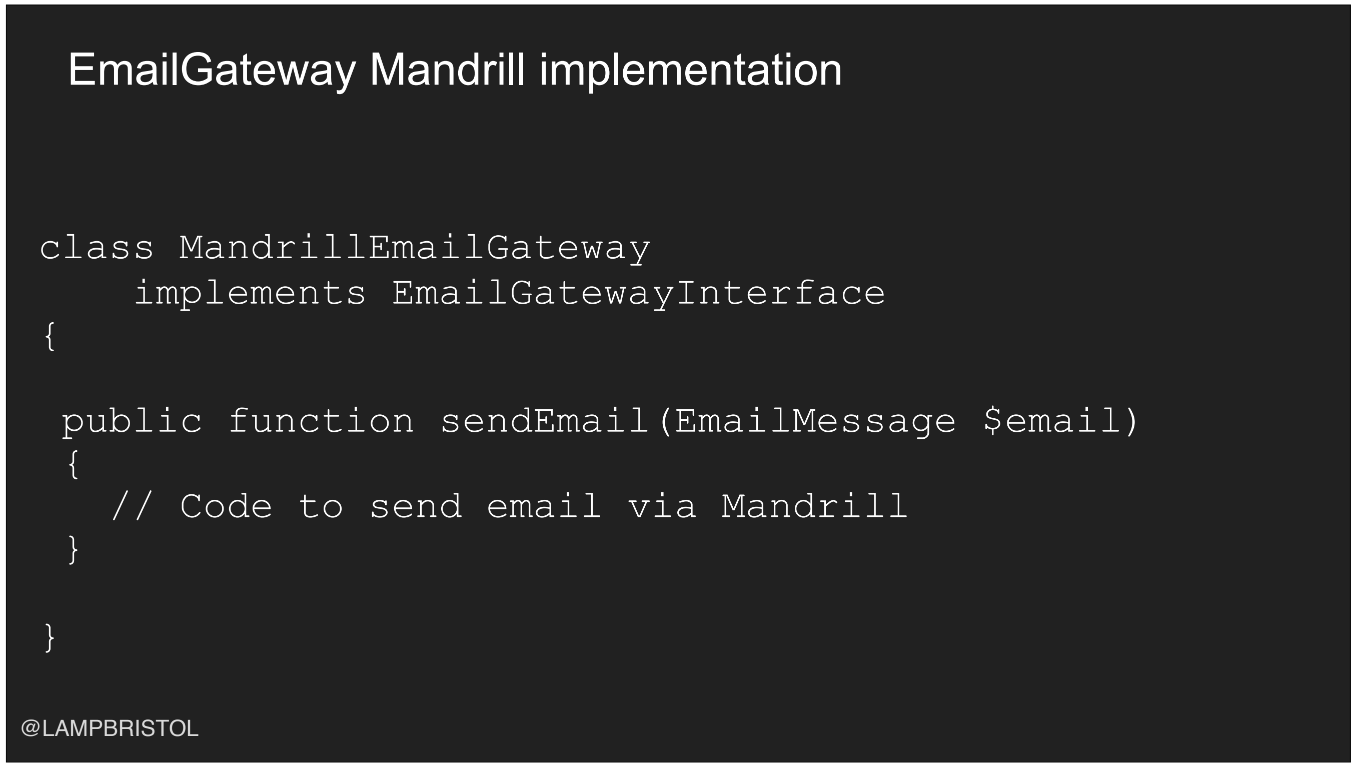
We want to save user in database. The finish the HTTP request and finally send the email without the client having to wait around.
We can hook into Symfony terminate event. This is called after the response has been sent to the HTTP request.
Client is waiting around for a fraction of the time they were previously.

EmailGateway

```
interface EmailGatewayInterface
{
  /**
   * Sends an email
   *
   */
  public function sendEmail(EmailMessage $email);
}
```

@LAMPBRISTOL

Reminder of our EmailGateway

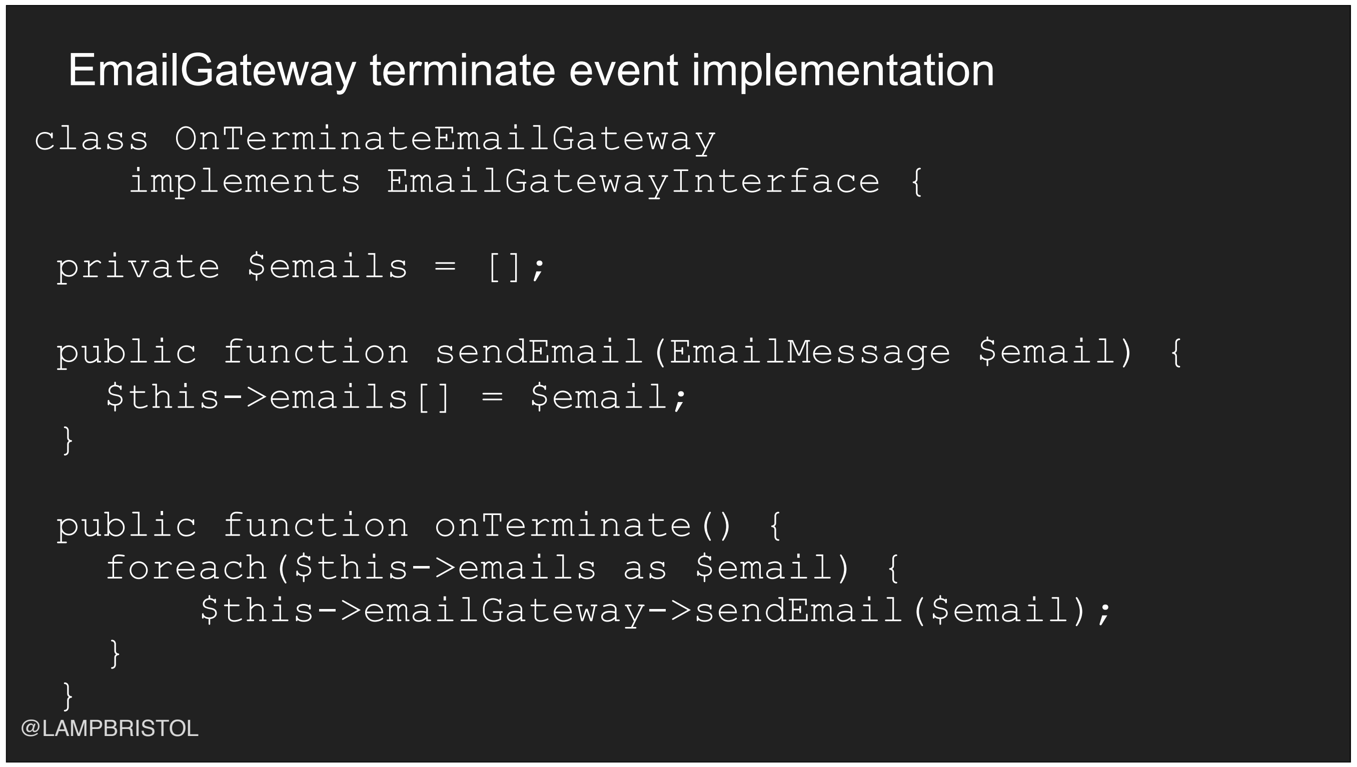


EmailGateway Mandrill implementation

```
class MandrillEmailGateway
    implements EmailGatewayInterface
{

 public function sendEmail(EmailMessage $email)
 {
   // Code to send email via Mandrill
 }

}
```

@LAMPBRISTOL

A possible implementation that actually sends the email.

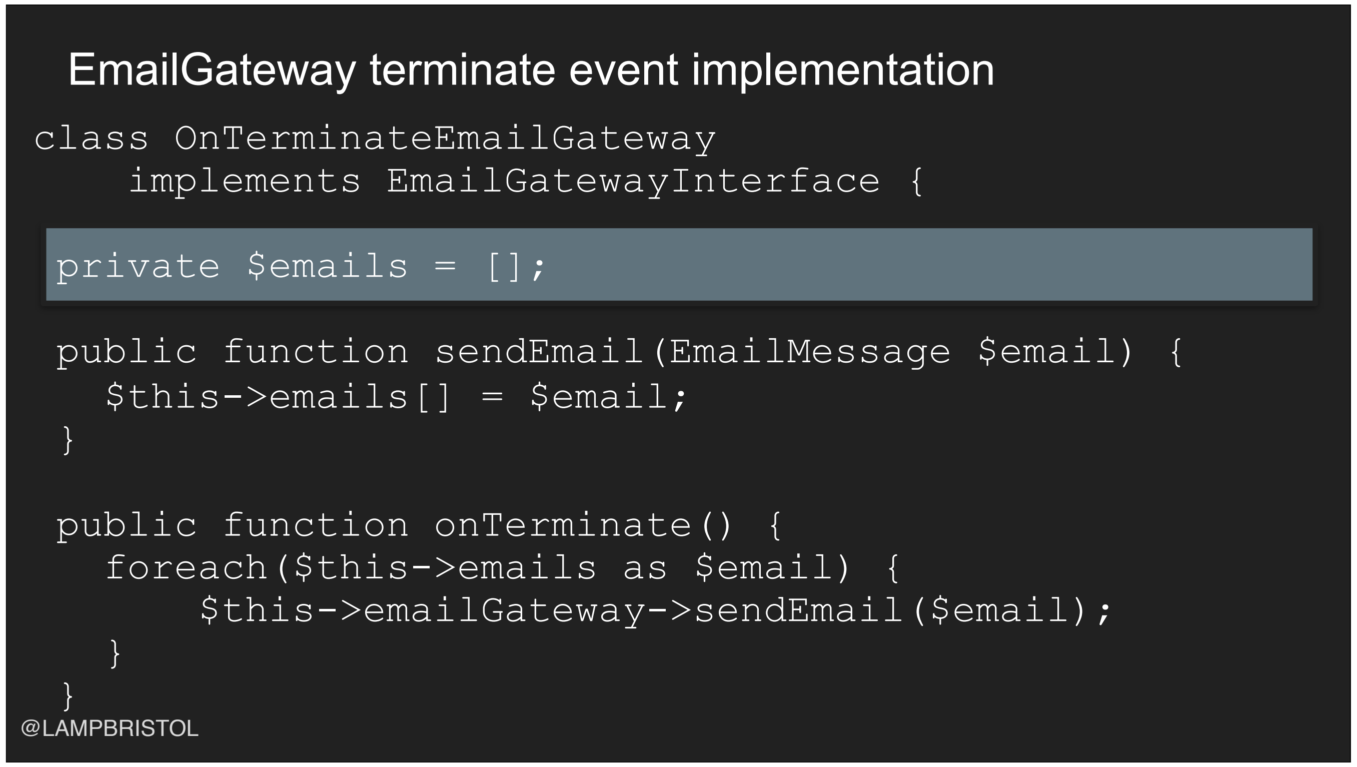


EmailGateway terminate event implementation

```
class OnTerminateEmailGateway
    implements EmailGatewayInterface {

 private $emails = [];

 public function sendEmail(EmailMessage $email) {
   $this->emails[] = $email;
 }

 public function onTerminate() {
   foreach($this->emails as $email) {
       $this->emailGateway->sendEmail($email);
   }
 }
```

@LAMPBRISTOL

Updated implementation for sending emails during the terminate phase of Symfony kernel.
- Keep a list of all emails to send
- Send email adds email to send to the list
- On terminate listener (called with the terminate event) loops through emails and actually sends them.

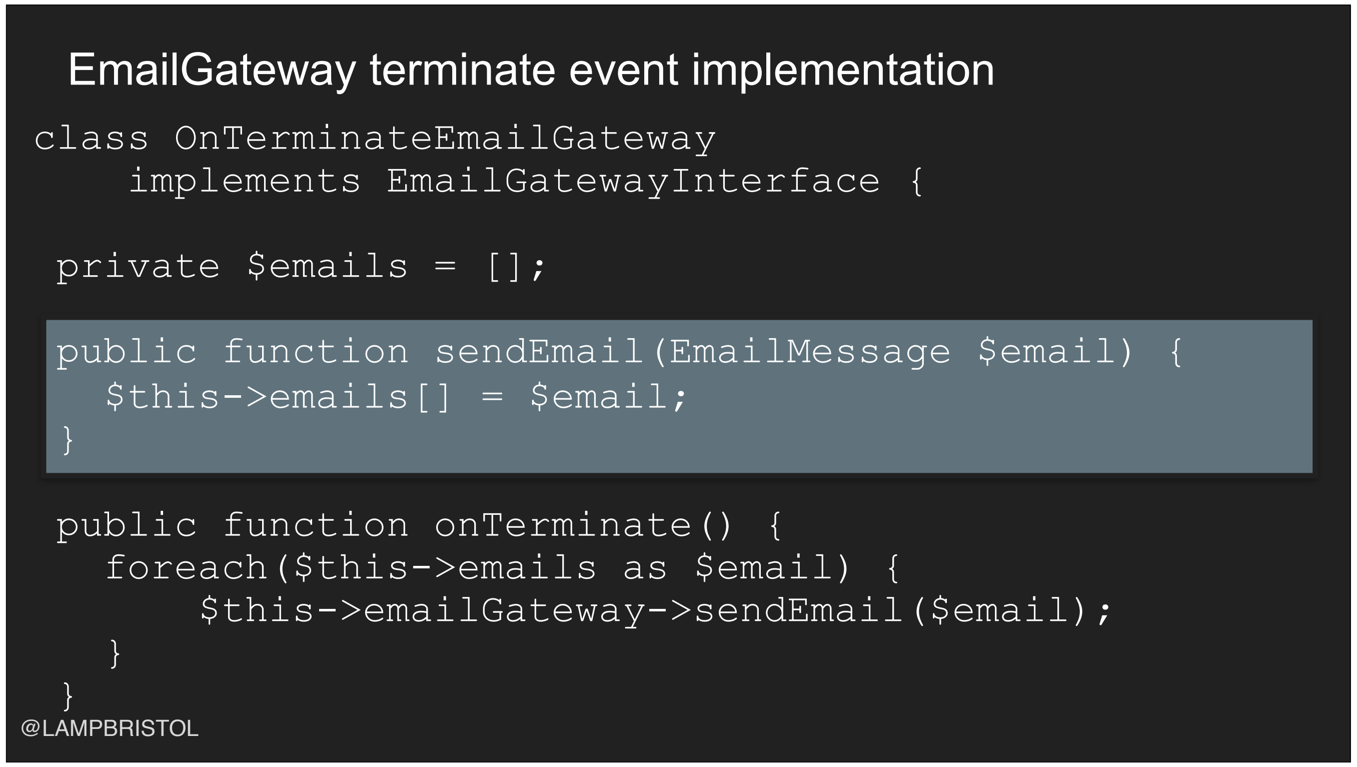


EmailGateway terminate event implementation

```
class OnTerminateEmailGateway
    implements EmailGatewayInterface {

  private $emails = [];

  public function sendEmail(EmailMessage $email) {
    $this->emails[] = $email;
  }

  public function onTerminate() {
    foreach($this->emails as $email) {
        $this->emailGateway->sendEmail($email);
    }
  }
```

@LAMPBRISTOL

Updated implementation for sending emails during the terminate phase of Symfony kernel.
- Keep a list of all emails to send
- Send email adds email to send to the list
- On terminate listener (called with the terminate event) loops through emails and actually sends them.

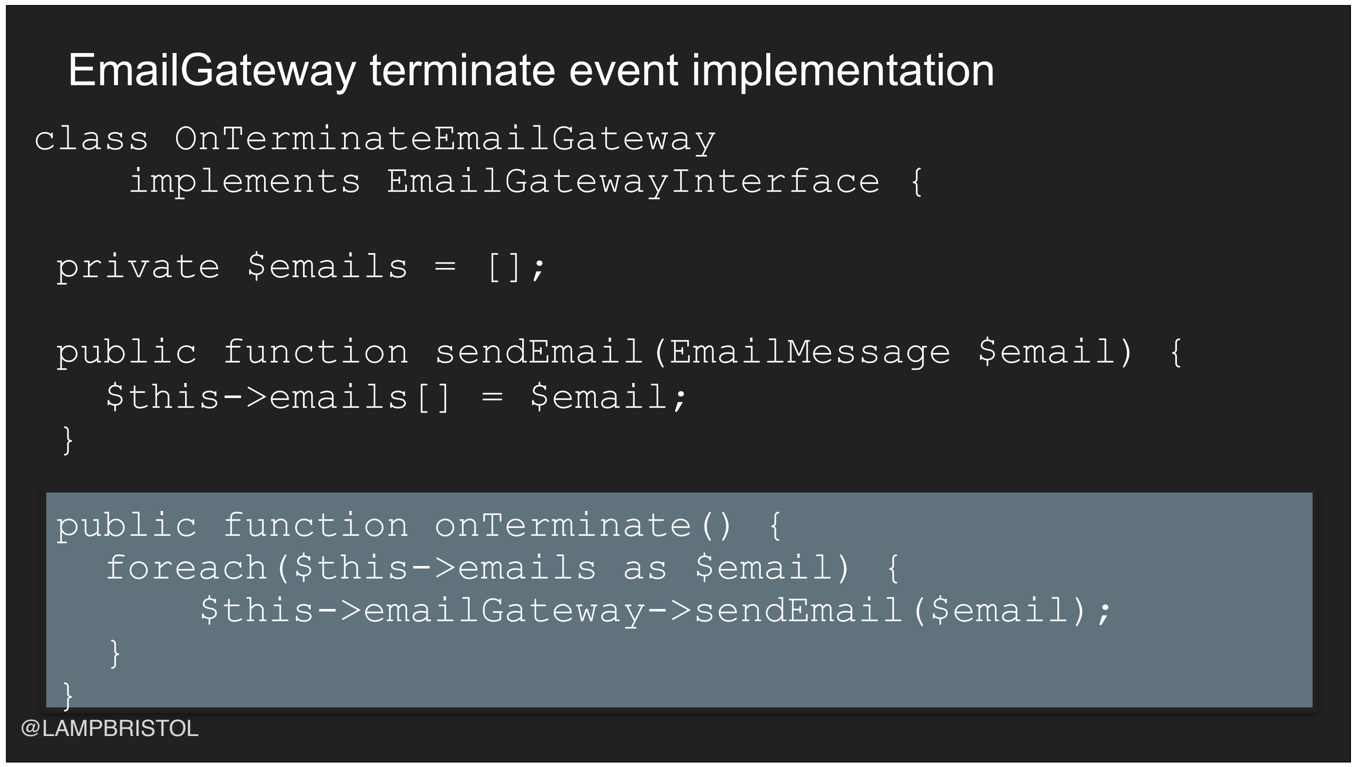


EmailGateway terminate event implementation

```
class OnTerminateEmailGateway
    implements EmailGatewayInterface {

 private $emails = [];

 public function sendEmail(EmailMessage $email) {
   $this->emails[] = $email;
 }

 public function onTerminate() {
   foreach($this->emails as $email) {
       $this->emailGateway->sendEmail($email);
   }
 }
```

@LAMPBRISTOL

Updated implementation for sending emails during the terminate phase of Symfony kernel.
- Keep a list of all emails to send
- Send email adds email to send to the list
- On terminate listener (called with the terminate event) loops through emails and actually sends them.



EmailGateway terminate event implementation

```
class OnTerminateEmailGateway
    implements EmailGatewayInterface {

 private $emails = [];

 public function sendEmail(EmailMessage $email) {
   $this->emails[] = $email;
 }

 public function onTerminate() {
   foreach($this->emails as $email) {
       $this->emailGateway->sendEmail($email);
   }
 }
```

@LAMPBRISTOL

Updated implementation for sending emails during the terminate phase of Symfony kernel.
- Keep a list of all emails to send
- Send email adds email to send to the list
- On terminate listener (called with the terminate event) loops through emails and actually sends them.