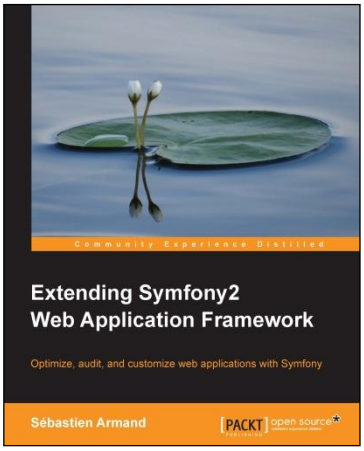
[PACKT] open source
community experience distilled
PUBLISHING

Extending Symfony2 Web Application Framework

Sébastien Armand



Chapter No. 1
"Services and Listeners"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Services and Listeners"

A synopsis of the book's content

Information on where to buy this book

About the Author

Sébastien Armand is a software developer based in Beijing, China. He spent most of the past five years working with Symfony, building internal IT systems. He co-founded `mashupsports.com`, a social website for sports enthusiasts based on Symfony2. He contributed to Symfony and the Symfony documentation on many occasions.

I would like to thank my ever-loving and understanding wife for all her support. If it weren't for her, I would have never started this book. Thank you. I'll be home for breakfast from now on! Also my parents and sister for just being awesome. Of course, I also extend my thanks to the whole Symfony community. It feels great being a part of it!

Extending Symfony2 Web Application Framework

The first stable version of Symfony2 was released more than two years back. Coming from all the experience acquired from Symfony1, the promise was to remove all the magic and provide a solid and modular basis to build web applications. The trade-off's inconvenience was justified in order for developers to regain full control and knowledge of the working of their application. To achieve this, it was decided that everything would be a bundle. The core framework itself is just a collection of bundles, which is everything you need to get started.

This great architecture being at the heart of Symfony2 and the promise of greater modularity and control over the whole framework enables any developer to create their own extensions. It is easy to implement these extensions; everything is prepared so that these extensions can be shared and their configuration can be disclosed for other developers to use them.

From the basics of creating a simple service to a custom authentication, this book will guide you through everything you need to create amazing bundles for Symfony2 and share them with the community.

What This Book Covers

Chapter 1, *Services and Listeners*, talks about the services and listeners that are the basis of nearly all extension techniques used in Symfony. This covers the fundamentals that will be reused throughout the book.

Chapter 2, *Commands and Templates*, helps you make your templates smarter and augment them with your own tailored functions and filters. This chapter helps you wrap common actions in commands so that you can perform them easily and reliably.

Chapter 3, *Forms*, helps you create your own form types and widgets and use them inside of dynamic forms that change based on the user information or even their own input.

Chapter 4, *Security*, discusses how to write custom authentication methods, use voters to restrict access, and add additional security layers to Symfony2.

Chapter 5, *Doctrine*, describes how to make your database fi t your data and not the opposite. This chapter also describes how to write custom database types and extend Doctrine to easily share common domain logic between models.

Chapter 6, *Sharing Your Extensions*, helps you to create a great extension that others could benefit from. It contains everything you need to know about publishing a self-contained reusable bundle.

1
Services and Listeners

This chapter will explain the basis of services in the Symfony2 framework. A **service** is an essential and core concept in Symfony2. In fact, most of the framework itself is just a big set of predefined services that are ready to use. As an example, if you just set up a new installation of Symfony2, from your project root, you can type `php app/console container:debug` to see the full list of services currently defined in your application. As you can see, even before we start writing anything for our application, we already have almost 200 services defined. The `php app/console container:debug <service_name>` command will provide information about a specific service and will be a useful command to refer to throughout the book.

Services

A service is just a specific instance of a given class. For example, whenever you access `doctrine` such as `$this->get('doctrine');` in a controller, it implies that you are accessing a service. This service is an instance of the `Doctrine EntityManager` class, but you never have to create this instance yourself. The code needed to create this entity manager is actually not that simple since it requires a connection to the database, some other configurations, and so on. Without this service already being defined, you would have to create this instance in your own code. Maybe you will have to repeat this initialization in each controller, thus making your application messier and harder to maintain.

Some of the default services present in Symfony2 are as follows:

- The annotation reader
- Assetic—the asset management library
- The event dispatcher
- The form widgets and form factory

- The Symfony2 Kernel and HttpKernel
- Monolog—the logging library
- The router
- Twig—the templating engine

It is very easy to create new services because of the Symfony2 framework. If we have a controller that has started to become quite messy with long code, a good way to refactor it and make it simpler will be to move some of the code to services. We have described all these services starting with "the" and a singular noun. This is because most of the time, services will be singleton objects where a single instance is needed.

A geolocation service

In this example, we imagine an application for listing events, which we will call "meetups". The controller makes it so that we can first retrieve the current user's IP address, use it as basic information to retrieve the user's location, and only display meetups within 50 kms of distance to the user's current location. Currently, the code is all set up in the controller. As it is, the controller is not actually that long yet, it has a single method and the whole class is around 50 lines of code. However, when you start to add more code, to only list the type of meetups that are the user's favorites or the ones they attended the most. When you want to mix that information and have complex calculations as to which meetups might be the most relevant to this specific user, the code could easily grow out of control!

There are many ways to refactor this simple example. The geocoding logic can just be put in a separate method for now, and this will be a good step, but let's plan for the future and move some of the logic to the services where it belongs. Our current code is as follows:

```
use Geocoder\HttpAdapter\CurlHttpAdapter;
use Geocoder\Geocoder;
use Geocoder\Provider\FreeGeoIpProvider;

public function indexAction()
  {
```

Initialize our geocoding tools (based on the excellent geocoding library at `http://geocoder-php.org/`) using the following code:

```
$adapter = new CurlHttpAdapter();
$geocoder = new Geocoder();
$geocoder->registerProviders(array(
  new FreeGeoIpProvider($adapter),
));
```

Retrieve our user's IP address using the following code:

```
$ip = $this->get('request')->getClientIp();
// Or use a default one
if ($ip == '127.0.0.1') {
  $ip = '114.247.144.250';
}
```

Get the coordinates and adapt them using the following code so that they are roughly a square of 50 kms on each side:

```
$result = $geocoder->geocode($ip);
$lat = $result->getLatitude();
$long = $result->getLongitude();
$lat_max = $lat + 0.25; // (Roughly 25km)
$lat_min = $lat - 0.25;
$long_max = $long + 0.3; // (Roughly 25km)
$long_min = $long - 0.3;
```

Create a query based on all this information using the following code:

```
$em = $this->getDoctrine()->getManager();
$qb = $em->createQueryBuilder();
$qb->select('e')
    ->from('KhepinBookBundle:Meetup, 'e')
    ->where('e.latitude < :lat_max')
    ->andWhere('e.latitude > :lat_min')
    ->andWhere('e.longitude < :long_max')
    ->andWhere('e.longitude > :long_min')
    ->setParameters([
      'lat_max' => $lat_max,
      'lat_min' => $lat_min,
      'long_max' => $long_max,
      'long_min' => $long_min
    ]);
```

Retrieve the results and pass them to the template using the following code:

```
    $meetups = $qb->getQuery()->execute();
    return ['ip' => $ip, 'result' => $result,
      'meetups' => $meetups];
}
```

The first thing we want to do is get rid of the geocoding initialization. It would be great to have all of this taken care of automatically and we would just access the geocoder with: `$this->get('geocoder');`.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can define your services directly in the `config.yml` file of Symfony under the `services` key, as follows:

```
services:
  geocoder:
    class: Geocoder\Geocoder
```

That is it! We defined a service that can now be accessed in any of our controllers. Our code now looks as follows:

```
// Create the geocoding class
$adapter = new \Geocoder\HttpAdapter\CurlHttpAdapter();
$geocoder = $this->get('geocoder');
$geocoder->registerProviders(array(
    new \Geocoder\Provider\FreeGeoIpProvider($adapter),
));
```

Well, I can see you rolling your eyes, thinking that it is not really helping so far. That's because initializing the geocoder is a bit more complex than just using the `new \Geocoder\Geocoder()` code. It needs another class to be instantiated and then passed as a parameter to a method. The good news is that we can do all of this in our service definition by modifying it as follows:

```
services:
    # Defines the adapter class
    geocoder_adapter:
        class: Geocoder\HttpAdapter\CurlHttpAdapter
        public: false
    # Defines the provider class
    geocoder_provider:
        class: Geocoder\Provider\FreeGeoIpProvider
        public: false
        # The provider class is passed the adapter as an argument
        arguments: [@geocoder_adapter]
    geocoder:
        class: Geocoder\Geocoder
```

```
# We call a method on the geocoder after initialization to
  set up the
# right parameters
calls:
    - [registerProviders, [[@geocoder_provider]]]
```

It's a bit longer than this, but it is the code that we never have to write anywhere else ever again. A few things to notice are as follows:

- We actually defined three services, as our geocoder requires two other classes to be instantiated.
- We used `@+service_name` to pass a reference to a service as an argument to another service.
- We can do more than just defining `new Class($argument);` we can also call a method on the class after it is instantiated. It is even possible to set properties directly when they are declared as `public`.
- We marked the first two services as `private`. This means that they won't be accessible in our controllers. They can, however, be used by the **Dependency Injection Container** (**DIC**) to be injected into other services.

Our code now looks as follows:

```
// Retrieve current user's IP address
$ip = $this->get('request')->getClientIp();

// Or use a default one
if ($ip == '127.0.0.1') {
    $ip = '114.247.144.250';
}
// Find the user's coordinates
$result = $this->get('geocoder')->geocode($ip);
$lat = $result->getLatitude();
// ... Remaining code is unchanged
```

Here, our controllers are extending the `BaseController` class, which has access to DIC since it implements the `ContainerAware` interface. All calls to `$this->get('service_name')` are proxied to the container that constructs (if needed) and returns the service.

Let's go one step further and define our own class that will directly get the user's IP address and return an array of maximum and minimum longitude and latitudes. We will create the following class:

```
namespace Khepin\BookBundle\Geo;

use Geocoder\Geocoder;
use Symfony\Component\HttpFoundation\Request;

class UserLocator {

    protected $geocoder;

    protected $user_ip;

    public function __construct(Geocoder $geocoder, Request
      $request) {
        $this->geocoder = $geocoder;
        $this->user_ip = $request->getClientIp();
        if ($this->user_ip == '127.0.0.1') {
            $this->user_ip = '114.247.144.250';
        }
    }

    public function getUserGeoBoundaries($precision = 0.3) {
        // Find the user's coordinates
        $result = $this->geocoder->geocode($this->user_ip);
        $lat = $result->getLatitude();
        $long = $result->getLongitude();
        $lat_max = $lat + 0.25; // (Roughly 25km)
        $lat_min = $lat - 0.25;
        $long_max = $long + 0.3; // (Roughly 25km)
        $long_min = $long - 0.3;
        return ['lat_max' => $lat_max, 'lat_min' => $lat_min,
           'long_max' => $long_max, 'long_min' => $long_min];
    }
}
```

It takes our `geocoder` and `request` variables as arguments, and then does all the heavy work we were doing in the controller at the beginning of the chapter. Just as we did before, we will define this class as a service, as follows, so that it becomes very easy to access from within the controllers:

```
# config.yml
services:
    #...
```

```
user_locator:
    class: Khepin\BookBundle\Geo\UserLocator
    scope: request
    arguments: [@geocoder, @request]
```

Notice that we have defined the scope here. The DIC has two scopes by default: `container` and `prototype`, to which the framework also adds a third one named `request`. The following table shows their differences:

Scope	Differences
Container	All calls to `$this->get('service_name')` return the same instance of the service.
Prototype	Each call to `$this->get('service_name')` returns a new instance of the service.
Request	Each call to `$this->get('service_name')` returns the same instance of the service within a request. Symfony can have subrequests (such as including a controller in Twig).

Now, the advantage is that the service knows everything it needs by itself, but it also becomes unusable in contexts where there are no requests. If we wanted to create a command that gets all users' last-connected IP address and sends them a newsletter of the meetups around them on the weekend, this design would prevent us from using the `Khepin\BookBundle\Geo\UserLocator` class to do so.

As we see, by default, the services are in the container scope, which means they will only be instantiated once and then reused, therefore implementing the singleton pattern. It is also important to note that the DIC does not create all the services immediately, but only on demand. If your code in a different controller never tries to access the `user_locator` service, then that service and all the other ones it depends on (`geocoder`, `geocoder_provider`, and `geocoder_adapter`) will never be created.

Also, remember that the configuration from the `config.yml` is cached when on a production environment, so there is also little to no overhead in defining these services.

Our controller looks a lot simpler now and is as follows:

```
$boundaries = $this->get('user_locator')->getUserGeoBoundaries();
// Create our database query
$em = $this->getDoctrine()->getManager();
$qb = $em->createQueryBuilder();
$qb->select('e')
    ->from('KhepinBookBundle:Meetup', 'e')
```

```
        ->where('e.latitude < :lat_max')
        ->andWhere('e.latitude > :lat_min')
        ->andWhere('e.longitude < :long_max')
        ->andWhere('e.longitude > :long_min')
        ->setParameters($boundaries);
    // Retrieve interesting meetups
    $meetups = $qb->getQuery()->execute();
    return ['meetups' => $meetups];
```

The longest part here is the doctrine query, which we could easily put on the repository class to further simplify our controller.

As we just saw, defining and creating services in Symfony2 is fairly easy and inexpensive. We created our own `UserLocator` class, made it a service, and saw that it can depend on our other services such as `@geocoder` service. We are not finished with services or the DIC as they are the underlying part of almost everything related to extending Symfony2. We will keep seeing them throughout this book; therefore, it is important to have a good understanding of them before continuing.

Testing services and testing with services

One of the great advantages of putting your code in a service is that a service is just a simple PHP class. This makes it very easy to unit test. You don't actually need the controller or the DIC. All you need is to create mocks of a `geocoder` and `request` class.

In the `test` folder of the bundle, we can add a `Geo` folder where we test our `UserLocator` class. Since we are only testing a simple PHP class, we don't need to use `WebTestCase`. The standard `PHPUnit_Framework_TestCase` will suffice. Our class has only one method that geocodes an IP address and returns a set of coordinates based on the required precision. We can mock the geocoder to return fixed numbers and therefore avoid a network call that would slow down our tests. A simple test case looks as follows:

```
class UserLocatorTest extends PHPUnit_Framework_TestCase
{
    public function testGetBoundaries()
    {
        $geocoder = $this->getMock('Geocoder\Geocoder');
        $result = $this->getMock('Geocoder\Result\Geocoded');

        $geocoder->expects($this->any())->method('geocode')-
          >will($this->returnValue($result));
        $result->expects($this->any())->method('getLatitude')-
          >will($this->returnValue(3));
```

```
            $result->expects($this->any())->method('getLongitude')
              ->will($this->returnValue(7));

            $request = $this->getMock
              ('Symfony\Component\HttpFoundation\Request',
                ['getUserIp']);
            $locator = new UserLocator($geocoder, $request);

            $boundaries = $locator->getUserGeoBoundaries(0);

            $this->assertTrue($boundaries['lat_min'] == 3);
        }
    }
```

We can now simply verify that our class itself is working, but what about the whole controller logic?

We can write a simple integration test for this controller and test for the presence and absence of some meetups on the rendered page. However, in some cases, for performance, convenience, or because it is simply not possible, we don't want to actually call the external services while testing. In that case, it is also possible to mock the services that will be used in the controller. In your tests, you will need to do the following:

```
public function testIndexMock()
{
    $client = static::createClient();
    $locator = $this->getMockBuilder
      ('Khepin\BookBundle\Geo\UserLocator')
        ->disableOriginalConstructor()->getMock();
    $boundaries = ["lat_max" => 40.2289, "lat_min" => 39.6289,
      "long_max" => 116.6883, "long_min" => 116.0883];
    $locator->expects($this->any())->method
      ('getUserGeoBoundaries')->will($this-
        >returnValue($boundaries));
    $client->getContainer()->set('user_locator', $locator);
    $crawler = $client->request('GET', '/');
  // Verify that the page contains the meetups we expect

}
```

Here, we mock the `UserLocator` class so that it will always return the same coordinates. This way, we can better control what we are testing and avoid waiting for a long call to the geolocation server.

Tagging services

You have most likely already encountered tagged services when using Symfony, for example, if you have defined custom form widgets or security voters. Event listeners, which we will talk about in the second part of this chapter, are also tagged services.

In our previous examples, we created a `user_locator` service that relies on a geocoder service. However, there are many possible ways to locate a user. We can have their address information in their profile, which will be faster and more accurate than getting it from a user's IP address. We can use different online providers such as FreeGeoIp as we did in the previous code, or have a local `geoip` database. We can even have all of these in our application at the same time, and try them one after the other from most to least accurate.

Let's define the interface for this new type of geocoder as follows:

```
namespace Khepin\BookBundle\Geo;

interface Geocoder
{
    public function getAccuracy();

    public function geocode($ip);
}
```

We will then define two geocoders using the following code; the first one just wraps our existing one in a new class that implements our `Geocoder` interface:

```
namespace Khepin\BookBundle\Geo;
use Geocoder\Geocoder as IpGeocoder;

class FreeGeoIpGeocoder implements Geocoder
{
    public function __construct(IpGeocoder $geocoder)
    {
        $this->geocoder = $geocoder;
    }

    public function geocode($ip)
    {
        return $this->geocoder->geocode($ip);
    }

    public function getAccuracy()
    {
        return 100;
    }
}
```

The first type of geocoder is configured as follows:

```
freegeoip_geocoder:
    class: Khepin\BookBundle\Geo\FreeGeoIpGeocoder
    arguments: [@geocoder]
```

The second geocoder returns a random location every time, as follows:

```
namespace Khepin\BookBundle\Geo;

class RandomLocationGeocoder implements Geocoder
{
    public function geocode($ip)
    {
        return new Result();
    }

    public function getAccuracy()
    {
        return 0;
    }
}

class Result
{
    public function getLatitude()
    {
        return rand(-85, 85);
    }

    public function getLongitude()
    {
        return rand(-180, 180);
    }

    public function getCountryCode()
    {
        return 'CN';
    }
}
```

The second geocoder is configured as follows:

```
random_geocoder:
    class: Khepin\BookBundle\Geo\RandomLocationGeocoder
```

Now, if we change the configuration of our `user_locator` service to use any of these geocoders, things will work correctly. However, what we really want is that it has access to all the available geolocation methods and then picks the most accurate one, even when we add new ones without changing the `user_locator` service.

Let's tag our services by modifying their configuration to add a tag as follows:

```
freegeoip_geocoder:
    class: Khepin\BookBundle\Geo\FreeGeoIpGeocoder
    arguments: [@geocoder]
    tags:
        - { name: khepin_book.geocoder }
random_geocoder:
    class: Khepin\BookBundle\Geo\RandomLocationGeocoder
    tags:
        - { name: khepin_book.geocoder }
```

We cannot pass all of these in the constructor of our class directly, so we'll modify our `UserLocator` class to have an `addGeocoder` method as follows:

```
class UserLocator
{    protected $geocoders = [];

    protected $user_ip;

    // Removed the geocoder from here
    public function __construct(Request $request)
    {
        $this->user_ip = $request->getClientIp();
    }

    public function addGeocoder(Geocoder $geocoder)
    {
        $this->geocoders[] = $geocoder;
    }

    // Picks the most accurate geocoder
    public function getBestGeocoder(){/* ... */}

    // ...
}
```

Informing the DIC that we want to add tagged services cannot be done only through configuration. This is instead done through a compiler pass when the DIC is being compiled.

Compiler passes allow you to dynamically modify service definitions. They can be used for tagged services and for creating bundles that enable extra functionalities whenever another bundle is also present and configured. The compiler pass can be used as follows:

```
namespace Khepin\BookBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler
  \CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class UserLocatorPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
      if (!$container->hasDefinition('khepin_book.user_locator'))
        {
            return;
        }

        $service_definition = $container->getDefinition
          ('khepin_book.user_locator');
        $tagged = $container->findTaggedServiceIds
        ('khepin_book.geocoder');

        foreach ($tagged as $id => $attrs) {
            $service_definition->addMethodCall(
              'addGeocoder',
              [new Reference($id)]
            );
        }
    }
}
```

After we have confirmed that the `user_locator` (renamed here as `khepin_book.user_locator`) service exists, we find all the services with the corresponding tag and modify the service definition for `khepin_book.user_locator` so that it loads these services.

You can define custom attributes on a tag. So, we could have moved the accuracy of each geocoder to the configuration as follows, and then used the compiler pass to only provide the most accurate geocoder to the user locator:

```
tags:
    - { name: khepin_book.geocoder, accuracy: 69 }
```

Whenever we define the YAML configuration for services, Symfony will internally create service definitions based on that information. By adding a compiler pass, we can modify these service definitions dynamically. The service definitions are then all cached so that we don't have to compile the container again.

Listeners

Listeners are a way of implementing the observer's design pattern. In this pattern, a particular piece of code does not try to start the execution of all the code that should happen at a given time. Instead, it notifies all of its observers that it has reached a given point in execution and lets these observers to take over the control flow if they have to.

In Symfony, we use the observer's pattern through events. Any class or function can trigger an event whenever it sees the event fit. The event itself can be defined in a class. This allows the passing of more information to the code observing this event. The framework itself will trigger events at different points in the process of handling the requests. These events are as follows:

- `kernel.request`: This event happens before reaching a controller. It is used internally to populate the `request` object.
- `kernel.controller`: This event happens immediately before executing the controller. It can be used to change the controller being executed.
- `kernel.view`: This event happens after executing the controller and if the controller did not return a `response` object. For example, this will be used to let Twig handle the rendering of a view by default.
- `kernel.response`: This event happens before the response is sent out. It can be used to modify the response before it is sent out.
- `kernel.terminate`: This event happens after the response has been sent out. It can be used to perform any time-consuming operations that are not necessary to generate the response.
- `kernel.exception`: This event happens whenever the framework catches an exception that was not handled.

Doctrine will also trigger events during an object's lifecycle (such as before or after persisting it to the database), but they are a whole different topic. You can learn everything about **Doctrine LifeCycle Events** at `http://docs.doctrine-project.org/en/latest/reference/events.html`.

Events are very powerful and we will use them in many places throughout this book. When you begin sharing your Symfony extensions with others, it is *always* a good idea to define and trigger custom events as these can be used as your own extension points.

We will build on the example provided in the *Services* section to see what use we could make of the listeners.

In the first part, we made our site that only shows a user the meetups that are happening around him or her. We now want to show meetups also taking into account a user's preferences (most joined meetups).

We have updated the schema to have a many-to-many relationship between users and the meetups as follows:

```
// Entity/User.php
/**
 * @ORM\ManyToMany(targetEntity="Meetup", mappedBy="attendees")
 */
protected $meetups;

// Entity/Meetup.php
/**
 * @ORM\ManyToMany(targetEntity="User", inversedBy="meetups")
 */
protected $attendees;
```

In our controller, we have a simple action to join a meetup, which is as follows:

```
/**
 * @Route("/meetups/{meetup_id}/join")
 * @Template()
 */
public function joinAction($meetup_id) {
    $em = $this->getDoctrine()->getManager();
    $meetup = $em->getRepository('KhepinBookBundle:Meetup')
      ->find($meetup_id);

    $form = $this->createForm(
      new JoinMeetupType(),
      $meetup,
      ['action' => '', 'method' => 'POST']
    );
    $form->add('submit', 'submit', array('label' => 'Join'));
```

```
        $form->handleRequest($this->get('request'));

        $user = $this->get('security.context')->getToken()->getUser();

        if ($form->isValid()) {
            $meetup->addAttendee($user);
            $em->flush();
        }

        $form = $form->createView();
        return ['meetup' => $meetup, 'user' => $user,
          'form' => $form];
    }
```

We use a form even for such a simple action because getting all our information from the URL in order to update the database and register this user as an attendee would enable many vulnerability issues such as CSRF attacks.

Updating user preferences using custom events

We want to add some code to generate the new list of favorite meetups of our user. This will allow us to change the logic for displaying the frontpage. Now, we can not only show users all the meetups happening around them, but also data will be filtered as to how likely they are to enjoy this kind of meetup. Our users will view the frontpage often, making the cost of calculating their favorite meetups on each page load very high. Therefore, we prefer to have a pre-calculated list of their favorite meetup types. We will update this list whenever a user joins or resigns from a meetup. In the future, we can also update it based on the pages they browse, even without actually joining the meetup.

The problem now is to decide where this code should live. The easy and immediate answer could be to add it right here in our controller. But, we can see that this logic doesn't really belong here. The controller makes sure that a user can join a meetup. It should limit its own logic to just doing that.

What is possible though is to let the controller call an event, warning all observers that a user has joined a meetup and letting these observers decide what is best to do with this information.

For this event to be useful, it needs to hold information about the user and the meetup. Let's create a simple class using the following code to hold that information:

```
// Bundle/Event/MeetupEvent.php
namespace Khepin\BookBundle\Event;

use Symfony\Component\EventDispatcher\Event;
use Khepin\BookBundle\Entity\User;
use Khepin\BookBundle\Entity\Meetup;

class MeetupEvent extends Event
{
    protected $user;
    protected $event;

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

    public function getUser() {
        return $this->user;
    }

    public function getMeetup() {
        return $this->meetup;
    }
}
```

This class is very simple and is only here to hold data about an event regarding a meetup and a user. Now let's trigger that event whenever a user joins a meetup. In our controller, use the following code after validating the form:

```
if ($form->isValid()) {
    $meetup->addAttendee($user);
    // This is the new line
    $this->get('event_dispatcher')->dispatch(
      'meetup.join',
      new MeetupEvent($user, $meetup)
    );
    $em->flush();
}
```

All we did was find the `event_dispatcher` service and dispatch the `meetup.join` event associated with some data. Dispatching an event is nothing more than just sending a message under a name, `meetup.join` in our case, potentially with some data. Before the code keeps on executing to the next line, all the classes and objects that listen to that event will be given the opportunity to run some code as well.

It is a good practice to namespace your events to avoid event name collisions. The dot (`.`) notation is usually preferred to separate event namespaces. So, it's very common to find events such as `acme.user.authentication.success`, `acme.user.authentication.fail`, and so on.

Another good practice is to catalog and document your events. We can see that if we keep on adding many events, since they are so easy to trigger because it's only a name, we will have a hard time keeping track of what events we have and what their purpose is. It is even more important to catalog your events if you intend to share your code with other people at some point. To do that, we create a static events class as follows:

```
namespace Khepin\BookBundle\Event;

final class MeetupEvents
{
    /**
     * The meetup.join event is triggered every time a user
     * registers for a meetup.
     *
     * Listeners receive an instance of:
     * Khepin\BookBundle\Event\MeetupEvent
     */
    const MEETUP_JOIN = 'meetup.join';
}
```

As we said, this class is much more for documentation purposes than anything else. Your code can now be changed in the controller as follows:

```
$container->get('event_dispatcher')->dispatch(
    MeetupEvents::MEETUP_JOIN,
    new MeetupEvent($user, $meetup)
);
```

We now know how to trigger an event, but we can't say that it has helped us to achieve anything interesting so far! Let's add a little bit of logic based on that. We will first create a listener class using the following code that will be responsible for generating the user's new list of preferred meetups:

```
namespace Khepin\BookBundle\Event\Listener;
use Khepin\BookBundle\Event\MeetupEvent;

class JoinMeetupListener
{
    public function generatePreferences(MeetupEvent $event) {
        $user = $event->getUser();
        $meetup = $event->getMeetup();
        // Logic to generate the user's new preferences
    }
}
```

Our class is a plain PHP class; it doesn't need to extend anything special. Therefore, it doesn't need to have any specific name. All it needs is to have at least one method that accepts a `MeetupEvent` argument. If we were to execute the code now, nothing would happen as we never said that this class should listen to a specific event. This is done by making this class a service again. This means that our listener could also be passed an instance of our geolocation service that we defined in the first part of this chapter, or any other existing Symfony service. The definition of our listener as a service, however, shows us some more advanced use of services:

```
join_meetup_listener:
    class: Khepin\BookBundle\Event\Listener\JoinMeetupListener
    tags:
        - { name: kernel.event_listener, event: meetup.join,
          method: generatePreferences }
```

What the `tags` section means is that when the `event_dispatcher` service is first created, it will also look for other services that were given a specific tag (`kernel.event_listener` in this case) and remember them. This is used by other Symfony components too, such as the form framework (which we'll see in *Chapter 3*, *Forms*).

Improving user performance

We have achieved something great by using events and listeners. All the logic related to calculating a user's meetup preferences is now isolated in its own listener class. We didn't detail the implementation of that logic, but we already know from this chapter that it would be a good idea to not keep it in the controller, but as an independent service that could be called from the listener. The more you use Symfony, the more this idea will seem clear and obvious; all the code that can be moved to a service should be moved to a service. Some Symfony core developers even advocate that controllers themselves should be services. Following this practice will make your code simpler and more testable.

Code that works after the response

Now, when our site grows in complexity and usage, our calculation of users' preferred event types could take quite a while. Maybe the users can now have friends on our site, and we want a user's choice to also affect his or her friend's preferences.

There are many cases in modern web applications where very long operations are not essential in order to return a response to the user. Some of the cases are as follows:

- After uploading a video, a user shouldn't wait until the conversion of the video to another format is finished before seeing a page that tells him or her that the upload was successful
- A few seconds could maybe be saved if we don't resize the user's profile picture before showing that the update went through
- In our case, the user shouldn't wait until we have propagated to all his or her friends the news of him or her joining a meetup, to see that he or she is now accepted and taking part in the meetup

There are many ways to deal with such situations and to remove unnecessary work from the process of generating a response. You can use batch processes that will recalculate all user preferences every day, but this will cause a lag in response time as the updates will be only once a day, and can be a waste of resources. You can also use a setup with a message queue and workers, where the queue notifies the workers that they should do something. This is somewhat similar to what we just did with events, but the code taking care of the calculation will now run in a different process, or maybe even on a different machine. Also, we won't wait for it to complete in order to proceed.

Symfony offers a simple way to achieve this while keeping everything inside the framework. By listening to the `kernel.terminate` event, we can run our listener's method after the response has been sent to the client.

We will update our code to take advantage of this. Our new listener will now behave as explained in the following table:

Event	Listener
`meetup.join`	Remembers the user and meetup involved for later use. No calculation happens.
`kernel.terminate`	Actually generates the user preferences. The heavy calculation takes place.

Our code should then look as follows:

```
class JoinMeetupListener
{
    protected $event;

    public function onUserJoinsMeetup(MeetupEvent $event) {
        $this->event = $event;
    }

    public function generatePreferences() {
        if ($this->event) {
            // Generate the new preferences for the user
        }
    }
}
```

We then need to also update the configuration to call `generatePreferences` on the `kernel.terminate` event, as follows:

```
join_meetup_listener:
        class: Khepin\BookBundle\Event\Listener\JoinMeetupListener
        tags:
            - { name: kernel.event_listener, event: meetup.join,
              method: onUserJoinsMeetup }
            - { name: kernel.event_listener, event:
              kernel.terminate, method: generatePreferences }
```

This is done very simply by only adding a tag to our existing listener. If you were thinking about creating a new service of the same class but listening on a different event, you will have two different instances of the service. So, the service that remembered the event will never be called to generate the preferences, and the service called to generate the preferences will never have an event to work with. Through this new setup, our heavy calculation code is now out of the way for sending a response to the user, and he or she can now enjoy a faster browsing experience.

Summary

This chapter introduced two of the most important concepts in Symfony, especially when it comes to extending the framework. By creating our geocoding service, we saw how easy it is to add a service that is just like any of the other Symfony services. We also reviewed how to use events to keep your code logic where it belongs and avoid cluttering your controllers with unwanted code. Then finally, we used them to make your site faster and more responsive to your users.

Believe it or not, if you really understand services and events, you know almost everything about extending Symfony. You will see throughout this book that we will constantly keep referring to both of these concepts, so, it is important that you have a good understanding of them.

In the next chapter, we will augment Symfony by adding new commands to the console tool and customize the templating engine. We will see that the services can be really helpful there as well.

Where to buy this book

You can buy Extending Symfony2 Web Application Framework from the Packt Publishing website: `http://www.packtpub.com/extending-symfony-2-web-application-framework/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/extending-symfony-2-web-application-framework/book