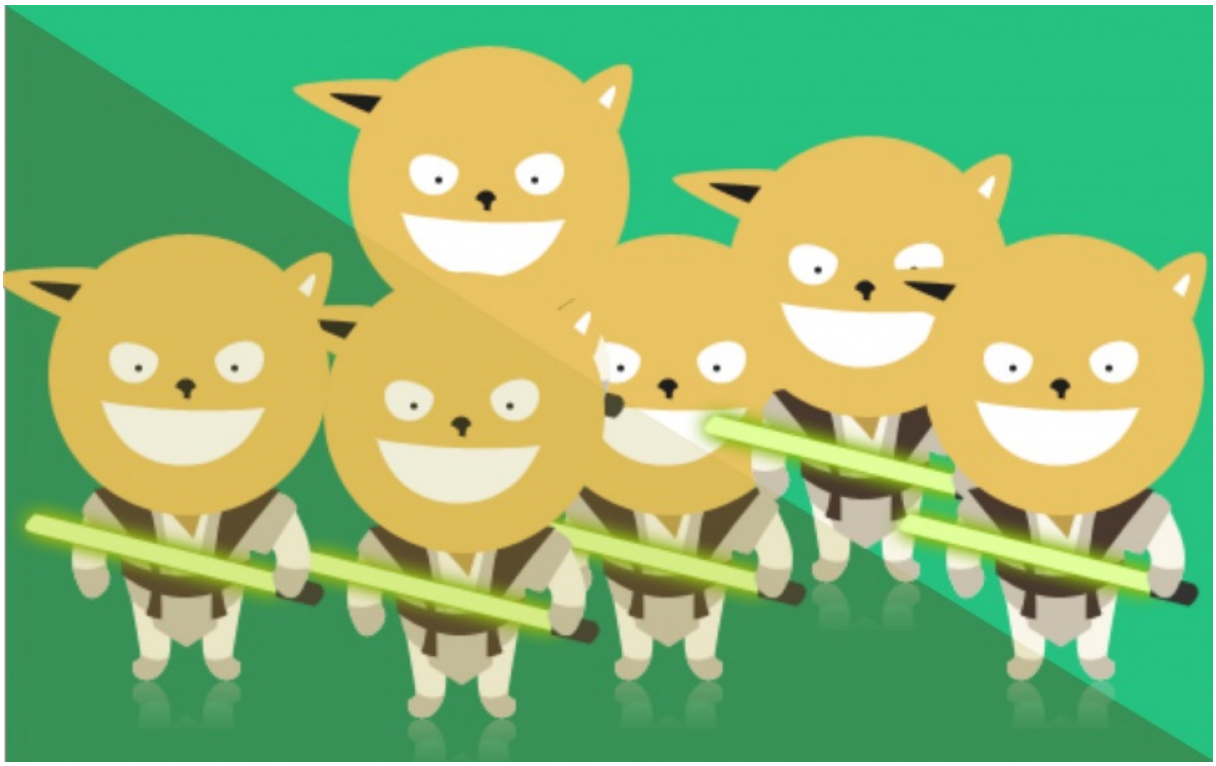
Starting in Symfony2: Course 3 (2.4+)



With <3 from SymfonyCasts

Chapter 1: Introduction

INTRODUCTION¶

Well hey friend! You've made it to part 3, so high-five a stranger and then keep going!

In this episode, I want us to really start to learn how Symfony works under the hood. We'll learn what a service is, find out more about the core Symfony services and create a few of our own. In Doctrine, we'll create some ManyToOne and ManyToMany relationships. We'll also talk about lifecycle callbacks and event listeners. And there's a lot more that we're hiding in between all of these topics.

And you have been coding with me, right? If you have you should now feel comfortable creating new classes, routes and templates. You're going to really kill that first project!

So when you see new concepts, stop the video, investigate, play with them, and then keep going. Ultimately, looking inside some of Symfony's own classes is a great way to get even more comfortable.

Ok let's roll!

Chapter 2: ManyToOne Doctrine Relationships

MANYTOONE DOCTRINE RELATIONSHIPS¶

Right now, if I creat an Event, there's no database link back to my user. We don't know which user created each Event.

To fix this, we need to create a `OneToMany` relationship from `User` to `Event`. In the database, this will mean a `user_id` foreign key column on the `yoda_event` table.

In Doctrine, relationships are handled by creating links between objects. Start by creating an `owner` property inside Event:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

class Event
{
    // ...

    protected $owner;
}
```

For normal fields, we use the `@ORM\Column` annotation. But for relationships, we use `@ORM\ManyToOne`, `@ORM\ManyToMany` or `@ORM\OneToMany`. This is a `ManyToOne` relationship because many events may have the same *one* `User`. I'll talk about the other 2 relationships later (*OneToMany*, *ManyToMany*).

Add the `@ORM\ManyToOne` relationship and pass in the entity that forms the other side:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToOne(targetEntity="Yoda\UserBundle\Entity\User")
 */
protected $owner;
```

Next, create the getter and setter for the the new property:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

use Yoda\UserBundle\Entity\User;

class Event
{
    // ...

    public function getOwner()
    {
        return $this->owner;
    }

    public function setOwner(User $owner)
    {
        $this->owner = $owner;
    }
}
```

Notice that when we call `setOwner`, we'll pass it an actual `User` object, *not* the id of a user. But when you save an `Event`, Doctrine will use the owner's id value to populate an `owner_id` column on the `yoda_event` table. So we link objects to objects in PHP, and Doctrine takes care of setting the foreign key id value for us. If you're newer to an ORM, this is one of the toughest things to understand about Doctrine.

Updating the Database¶

How can we update our database with the new column and foreign key? Why, with the `doctrine:schema:update` command of course! I'll dump the SQL to the terminal first to see it:

```
php app/console doctrine:schema:update --dump-sql
php app/console doctrine:schema:update --force
```

As expected, the SQL that's generated will add a new `owner_id` field to `yoda_event` along with the foreign key constraint.

ManyToOne Options¶

Since I'm feeling fancy, let's configure a few things. Whenever you have a `ManyToOne` annotation, you can optionally add an `@ORM\JoinColumn` annotation to control some database options.

JoinColumn onDelete¶

To add a database-level "ON DELETE" cascade behavior, add the `onDelete` option:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToOne(targetEntity="Yoda\UserBundle\Entity\User")
 * @ORM\JoinColumn(onDelete="CASCADE")
 */
protected $owner;
```

Now, let's run the `doctrine:schema:update` command again:

```
php app/console doctrine:schema:update --dump-sql
php app/console doctrine:schema:update --force
```

The SQL tells us that this actually re-creates the foreign key with the "on delete" behavior. So if we delete a `User`, the database will automatically delete all rows in the `yoda_event` table that link to that user and ship them off into hyper space.

The cascade Option¶

Another common option is `cascade` on the actual `ManyToOne` part:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToOne(targetEntity="Yoda\UserBundle\Entity\User", cascade={"remove"})
 * @ORM\JoinColumn(onDelete="CASCADE")
 */
protected $owner;
```

This is like `onDelete`, but in the opposite direction. With this, if we delete an Event, it will *cascade* the remove onto the owner. In other words, If I delete an Event, it will also delete the User who is the owner.

Run `doctrine:schema:update` again:

```
php app/console doctrine:schema:update --dump-sql
```

Now, it doesn't want to change our database at all. Unlike `onDelete`, this behavior is enforced entirely by Doctrine in PHP, not in the database layer.

Tip

You can also cascade `persist`, which is useful at times with `ManyToMany` relationship where you're creating new items in the relationship.

Remove the `cascade` option because it's dangerous in our situation:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToOne(targetEntity="Yoda\UserBundle\Entity\User")
 * @ORM\JoinColumn(onDelete="CASCADE")
 */
protected $owner;
```

If we delete an Event, we definitely don't want that to delete the Event's owner. Darth would be so angry.

Linking an Event to its owner on creation¶

Time to put our shiny relationship to the test. When a new `Event` object is created, let's associate it with the `User` object for whoever is logged in:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function createAction(Request $request)
{
    // ...

    if ($form->isValid()) {
        $user = $this->getUser();

        // ...
    }
}
```

To complete the link, just call `setOwner` on the Event and pass in the *whole* `User` object:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function createAction(Request $request)
{
    // ...

    if ($form->isValid()) {
        $user = $this->getUser();

        $entity->setOwner($user);

        // ... the existing save logic
    }
}
```

Yep, that's it. When we save the Event, Doctrine will automatically grab the id of the `User` object and place it on the `owner_id` field.

Time to test! Login as Wayne. Remember, he has `ROLE_ADMIN`, which also means he has `ROLE_EVENT_CREATE` because of the `role_hierarchy` section in `security.yml`.

Now, fill in some basic data and submit it. To see the result, use the query tool to list the events:

```
php app/console doctrine:query:sql "SELECT * FROM yoda_event"
```

Sure enough, our newest event is linked back to our user! #Winning

Chapter 3: Sharing Data between Fixture Classes

SHARING DATA BETWEEN FIXTURE CLASSES¶

Let's update the fixtures so that each event has an owner.

We have two fixture classes: one that loads events and one that loads users.

Ordering how Fixtures are Loaded¶

Start in the `LoadUsers` class. Now that events depend on users, we'll want this fixture class to be executed *before* the events class. To force this, implement a new interface called `OrderedFixtureInterface`. This requires one method called `getOrder`. Let's return 10:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

use Doctrine\Common\DataFixtures\OrderedFixtureInterface;

class LoadUsers implements FixtureInterface, ContainerAwareInterface, OrderedFixtureInterface
{
    // ...

    public function getOrder()
    {
        return 10;
    }
}
```

Head over to `LoadEvents` and make the same change, except returning 20 so that the class is run second:

```
// src/Yoda/EventBundle/DataFixtures/ORM/LoadEvents.php
// ...

use Doctrine\Common\DataFixtures\OrderedFixtureInterface;

class LoadEvents implements FixtureInterface, OrderedFixtureInterface
{
    // ...

    public function getOrder()
    {
        return 20;
    }
}
```

Assigning Owners in Fixtures¶

Now, we just need to get our new User objects inside `LoadEvents`. DoctrineFixturesBundle has a standard way of sharing data between fixtures, but a much easier way is just to query for our wayne user:

```
// src/Yoda/EventBundle/DataFixtures/ORM/LoadEvents.php
// ...

class LoadEvents implements FixtureInterface, OrderedFixtureInterface
{
    $wayne = $manager->getRepository('UserBundle:User')
        ->findOneByUsernameOrEmail('wayne');

    // ...
}
```

All we need to do now is call `setOwner` on both events so that it looks like wayne created them:

```
// src/Yoda/EventBundle/DataFixtures/ORM/LoadEvents.php
// ...
public function load(ObjectManager $manager)
{
    $wayne = $manager->getRepository('UserBundle:User')
        ->findOneByUsernameOrEmail('wayne');
    // ...

    $event1->setOwner($wayne);
    $event2->setOwner($wayne);

    // ...
    $manager->flush();
}
```

Ok! Reload the fixtures!

```
php app/console doctrine:fixtures:load
```

Now use `app/console` to check that each event has an owner:

```
php app/console doctrine:query:sql "SELECT * FROM yoda_event"
```

Chapter 4: Restricting Edit Access to Owners

RESTRICTING EDIT ACCESS TO OWNERS¶

Now that every `Event` has an owner, let's prevent that meddling Darth from editing any events that he didn't create.

This should be pretty easy. If the current logged in `User` object doesn't match the Event's owner, we'll just deny access. And remember, you can deny access anywhere in your app just by throwing the special `AccessDeniedException`.

Since we'll need the same security logic in `editAction`, `updateAction` and `deleteAction`, let's create a private function called `enforceOwnerSecurity` that holds it:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

private function enforceOwnerSecurity(Event $event)
{
    $user = $this->getUser();

    if ($user != $event->getOwner()) {
        // if you're using 2.5 or higher
        // throw $this->createAccessDeniedException('You are not the owner!!!');
        throw new AccessDeniedException('You are not the owner!!!');
    }
}
```

It's now pretty simple to prevent Darth from doing things with events he didn't create. Just call this function from `editAction`, `updateAction` and `deleteAction`:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function editAction($id)
{
    // ...

    if (!$entity) {
        throw $this->createNotFoundException('Unable to find Event entity.');
    }

    $this->enforceOwnerSecurity($entity);
    // ...
}

// repeate for updateAction and deleteAction
```

Ok, log in as Darth and try to edit an event. Denied!

In the production environment, the user will see a 403 page that you can customize. And in a few minutes, we'll show you *how*.

Tip

There is an even cleaner, but more advanced, approach to restricting access to specific objects called "voters". You can learn more about these from our Question and Answer Day. An even more advanced approach is available called ACLs.

Now that Darth can only edit an event if he created it, add an `if` statement around the edit link that hides it for all other users:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

{% if app.user == entity.owner %}
    <a class="button" href="{{ path('event_edit', {'id': entity.id}) }}">edit</a>
{% endif %}
```

Remember that this works because `app.user` gives us the `User` object for whoever is logged in.

Chapter 5: Using a shortcut Base Controller Class

USING A SHORTCUT BASE CONTROLLER CLASS¶

Getting the `security.context` service requires too much typing. So let's make some improvements so we can get things done faster.

Create a new class called `Controller` inside the `EventBundle` and make this class extend Symfony's standard base controller. But watch out! Both classes are called `Controller`, so we need to alias Symfony's class to `BaseController`:

```
// src/Yoda/EventBundle/Controller/Controller.php

namespace Yoda\EventBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller as BaseController;

class Controller extends BaseController
{
    // ...
}
```

Inside this class, create a function that returns the security context from the service container:

```
// src/Yoda/EventBundle/Controller/Controller.php
// ...

public function getSecurityContext()
{
    return $this->container->get('security.context');
}
```

Using the Custom Base Controller¶

Head back to `EventController`. Right now, this extends Symfony's Controller, which means that we get access to all of its shortcuts. Remove the `use` statement for Symfony's Controller and replace it with a `use` statement for *our* fancy Controller class:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

use Yoda\EventBundle\Controller\Controller;

class EventController extends Controller
{
    // ...
}
```

Now we can access all of Symfony's shortcut methods *and* the new `getSecurityContext` method we created. And actually, we don't even need the `use` statement because this class lives in the same namespace as the new `Controller` class.

Ok! Let's use the new `getSecurityContext` method to shorten things:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

private function enforceUserSecurity($role = 'ROLE_USER')
{
    if (!$this->getSecurityContext()->isGranted($role)) {
        // in Symfony 2.5
        // throw $this->createAccessDeniedException('message!');
        throw new AccessDeniedException('Need '.$role);
    }
}
```

And even though we're not really using its page, remove the `use` statement in `DefaultController` as well so that we're using the new class:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
// ...

// no use statement here anymore

class DefaultController extends Controller
{
// ...
```

Change the `use` statements in both `RegisterController` and `SecurityController`. In `RegisterController`, we can also take advantage of the new shortcut:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

use Yoda\EventBundle\Controller\Controller;

class RegisterController extends Controller
{
    // ...

    private function authenticateUser(User $user)
    {
        $providerKey = 'secured_area'; // your firewall name
        $token = new UsernamePasswordToken($user, null, $providerKey, $user->getRoles());

        $this->getSecurityContext()->setToken($token);
    }
}
```

```
// src/Yoda/UserBundle/Controller/SecurityController.php
// ...

use Yoda\EventBundle\Controller\Controller;
// ...

class SecurityController extends Controller
```

These controllers *do* need to have a `use` statement, because they don't live in the same namespace as the new `Controller` class.

Add More Methods to Controller!¶

Now that all of our controllers extend *our* Controller class, we can add whatever shortcut functions we want here. For example, if we needed to check for `Event` owner security in another controller, we could just move that function into

`Controller` and make it public:

```
// src/Yoda/EventBundle/Controller/Controller.php
// ...

use Yoda\EventBundle\Entity\Event;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

class Controller extends BaseController
{
    // ...

    public function enforceOwnerSecurity(Event $event)
    {
        $user = $this->getUser();

        if ($user != $event->getOwner()) {
            // if you're using 2.5 or higher
            // throw $this->createAccessDeniedException('You are not the owner!!!');
            throw new AccessDeniedException('You are not the owner!!!');
        }
    }
}
```

Chapter 6: Using PHPDoc for Auto-Completion

USING PHPDOC FOR AUTO-COMPLETION¶

With the base Controller, we can give ourselves shortcuts to develop faster and faster.

Inside `RegisterController`, my IDE recognizes the `setToken` method on the security context automatically. Actually, this only works because I'm using an awesome Symfony2 plugin for PHPStorm. The `getSecurityContext` method doesn't have any PHPDoc, so any other editor will have no idea what type of object this method returns.

To fix this, and because PHPDoc is a good practice, let's add some to our new method:

```
// src/Yoda/EventBundle/Controller/Controller.php
// ...

/**
 * @return \Symfony\Component\Security\Core\SecurityContext
 */
public function getSecurityContext()
{
    return $this->container->get('security.context');
}
```

Because of the Symfony2 plugin, the `@return` tag was filled in automatically. That's awesome! But if it hadn't, we could figure out what type of object `security.context` is by using the `container:debug` console command:

```
php app/console container:debug security.context
```

If you use PHPStorm, install the Symfony Plugin. If not, rely on this console command to help you find out more about a service.

Re-Running the Tests¶

It's like you read my mind! Now is a prefect time to re-run the test suite to make sure we haven't broken anything. I know I know, we're missing tests for some important parts, like event creation, but it's better than nothing.

But first, update your test database for our latest schema changes:

```
php app/console doctrine:schema:update --force --env=test
```

We need this because we configured our project in episode 2 to use an entirely different database for testing.

```
./bin/phpunit -c app/
```

Chapter 7: OneToMany: The Inverse Side of a Relationship

ONETOMANY: THE INVERSE SIDE OF A RELATIONSHIP¶

Earlier, we gave every `Event` an owner. This was our first Doctrine relationship: a `ManyToOne` from `Event` to `User`.

This lets us do things like call `$event->getOwner()`. Let's use this to print the owner of an `Event`:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

{{ entity.owner.username }}
```

But what about the opposite direction? Can we start with a `$user` object and call `getEvents()` to get all the `Event` objects the User has created?

Trying User::getEvents()¶

Open up the play script we made in episode one to test this out. Clear out all the code below the setup, then query for a `User` object and call `getEvents()` on it:

```
// play.php
// ...
// all our setup is done!!!!!!

$em = $container->get('doctrine')->getManager();

$user = $em
    ->getRepository('UserBundle:User')
    ->findOneBy(array('username' => 'wayne'))
;

foreach ($user->getEvents() as $event) {
    var_dump($event->getName());
}
```

Now run the script:

```
php play.php
```

It blows up!

> Call to undefined method YodaUserBundleEntityUser::getEvents()

This shouldn't surprise us. The `User` object is a plain PHP object and we've never added a `getEvents` method to it.

Setting up User::getEvents()¶

We *can* do this, and it's not hard, but it can be tricky to understand. It involves 3 steps.

Step 1: Add the OneToMany annotation¶

Start by adding an `events` property to `User`. Give it a `OneToMany` annotation:

```php
// src/Yoda/UserBundle/Entity/User.php
// ...

/**
 * @ORM\OneToMany(targetEntity="Yoda\EventBundle\Entity\Event", mappedBy="owner")
 */
protected $events;
```

This looks just like the `ManyToOne` annotation we used inside `Event`, except for the extra `mappedBy` property, which tells Doctrine which property inside `Event` this maps to.

Step 2: Add inversedBy to ManyToOne¶

Second, now that we have the `OneToMany`, you also need to go to `Event` and add an `inversedBy` option pointing back to the `events` property on `User`:

```php
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToOne(
 *     targetEntity="Yoda\UserBundle\Entity\User",
 *     inversedBy="events"
 * )
 * @ORM\JoinColumn(onDelete="CASCADE")
 */
protected $owner;
```

I broke this onto multiple lines only to make things more readable.

Step 3: Initializing the ArrayCollection¶

Finally, in `User`, create a `__construct` method and set the `events` property to a special `ArrayCollection` object:

```php
// src/Yoda/UserBundle/Entity/User.php
// ...
use Doctrine\Common\Collections\ArrayCollection;

public function __construct()
{
    $this->events = new ArrayCollection();
}
```

In a perfect world, the `events` property would just be an array of `Event` objects. But for Doctrine to work its magic, we need it to be an `ArrayCollection` object instead. But no worries, this object looks and feels just like an array, so just think of it like one.

Complete things by adding the getter and setter for the the `events` property:

```php
// src/Yoda/UserBundle/Entity/User.php
// ..

public function getEvents()
{
    return $this->events;
}

public function setEvents(ArrayCollection $events)
{
    $this->events = $events;
}
```

Now try the play script:

```
php play.php
```

It works! And we see both event names, since wayne owns both of them.

Behind the scenes, Doctrine automatically queries for the two event objects owned by this wayne dude and puts them on the `events` property.

Owning Versus Inverse Side¶

Notice that we didn't have to make any database schema changes for this to work. That's really important. because adding this side of the relationship is purely for convenience. Our database already has all the information it needs to link `Users` and `Events`.

The `OneToMany` side of a relationship is always optional, and called the "inverse" side. If you need the convenience, add it. If you don't, don't bother with it.

The `ManyToOne` side of the relationship is where the foreign key actually lives in the database, and it's known as the "owning" side. You'll *always* need to specify the owning side of a relationship.

Caution: Don't "set" the Inverse Side¶

The inverse side is special for another important reason. If we called `setEvents()` on a `User` and saved, the new events would be ignored. Only the "owning" side of the relationship is used when saving.

For example, in `createAction` of `EventController`, we're currently calling `setOwner` on Event:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

// this works
$entity->setOwner($this->getUser());
```

This is perfect because `owner`, coincidentally, is the *owning* side of the relationship. In a `ManyToOne` and `OneToMany` association, the *owning* side is always the singular side. We are talking about *one* owner, so it's the owning side.

If instead we decided to call `setEvents()` on the `User`, we'd be setting the inverse side, and Doctrine would completely ignore it:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

// this does nothing
// if we *only* had this part, the relationship would not save
// $events = $this->getUser()->getEvents();
// $events[] = $entity;
// $this->getUser()->setEvents($events);
```

In fact, let's just remove `setEvents` from `User`, so that nobody calls this method on accident:

```
// src/Yoda/UserBundle/Entity/User.php
// ..

public function getEvents()
{
    return $this->events;
}

// setEvents() has been removed
```

The problem of not being able to set the relationship from both sides can be particularly tricky when working with a form that embeds many sub-forms. If you run into this, check out the cookbook entry on the topic at symfony.com. Also check out the reference manual for the collection form type.

Chapter 8: Doctrine Extensions: Sluggable and Timestampable

DOCTRINE EXTENSIONS: SLUGGABLE AND TIMESTAMPABLE¶

I want to show you a little bit of Doctrine magic by using an open source library called DoctrineExtensions. The first bit of magic we'll add is a `slug` to Event. Not the jabba the hutt variety, but a property that is automatically cleaned and populated based on the event name.

Installing the StofDoctrineExtensionsBundle¶

Head over to knpbundles.com and search for `doctrine extension`. The StofDoctrineExtensionsBundle is what we want: it brings in that DoctrineExtensions library and adds some Symfony glue to make things really easy. Click into its documentation.

Installing a bundle is always the same 3 steps. First, use Composer's `require` command and pass it the name of the library:

```
php composer.phar require stof/doctrine-extensions-bundle
```

If it asks you for a version, type `~1.1.0`. In the future, Composer should decide the best version for you.

Like we've seen before, the `require` command just added the library to `composer.json` for us and started downloading it.

Second, add the new bundle to your `AppKernel`:

```
// app/AppKernel.php
// ...

public function registerBundles()
{
    $bundles = array(
        // ...
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
    );

    // ...
}
```

And third, configure the bundle by copying a few lines from the README:

```
# app/config/config.yml
# ...

stof_doctrine_extensions:
    orm:
        default:    ~
```

All of the details on how to install a bundle and configure it will always live in its documentation.

Adding Sluggable to Event¶

This bundle brings in a bunch of cool features, which we have to activate manually in `config.yml`. The first is called "sluggable":

```
# app/config/config.yml
# ...

stof_doctrine_extensions:
    orm:
        default:
            sluggable:   true
```

Open up the `Event` entity and add a new property called `slug`:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\Column(length=255, unique=true)
 */
protected $slug;
```

This is just a normal property that will store a URL-safe and unique version of the event's name. And now let's add the getter and setter:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

public function getSlug()
{
    return $this->slug;
}

public function setSlug($slug)
{
    $this->slug = $slug;
}
```

Configuring slug to be set Automatically¶

Ready for the magic? Let's see if we can get the `slug` field to be automatically populated for us, based on the event's name.

The `StofDoctrineExtensionBundle` is actually just a wrapper around another library called `DoctrineExtensions` that does most of the work. We can go to its README to get real usage details. Find the `sluggable` section and look at the first example.

This library works via annotations, so copy and paste the new `use` statement into `Event`. Next, copy the annotation from the slug field and change the fields option to only include `name`:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

use Gedmo\Mapping\Annotation as Gedmo;
// ...

class Event
{
    // ...

    /**
     * @Gedmo\Slug(fields={"name"}, updatable=false)
     * @ORM\Column(length=255, unique=true)
     */
    protected $slug;
}
```

This says that we want DoctrineExtensions to automatically set the `slug` field based on the `name` property. If we also set `updatable` to `false`, it tells the library to set `slug` once and never change it again, even if the event's name changes. That's good because the slug will be used in the event's URL. And changing URLs is lame :).

Let's try it! Update the database schema:

```
php app/console doctrine:schema:update --force
```

This explodes because our existing events will all temporarily have blank slugs, which isn't unique. Drop the schema and rebuild from scratch to get around this:

```
php app/console doctrine:schema:drop --force
php app/console doctrine:schema:create
php app/console doctrine:fixtures:load
```

Reload the fixtures and check the results by querying for events via the console:

```
php app/console doctrine:query:sql "SELECT * FROM yoda_event"
```

Hey, we have slugs! That's not something you would be excited about outside of programming. As an added bonus, if two events have the same name, the library will automatically add a `-1` to the end of the second slug. The library has our back and makes sure that these are always unique.

Chapter 9: Using the slug in the Event URL

USING THE SLUG IN THE EVENT URL¶

We've got slugs! So let's enjoy them by putting them into our URLs!

First, change the `event_show` route to use the `slug` instead of the `id`:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
# ...

event_show:
    pattern:  /{slug}/show
    defaults: { _controller: "EventBundle:Event:show" }

# ...
```

You can also update the other routes if you want to - but this is the most important URL to get right.

Update the `showAction` accordingly and query for the `Event` using the slug:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function showAction($slug)
{
    $em = $this->getDoctrine()->getManager();

    $entity = $em->getRepository('EventBundle:Event')
        ->findOneBy(array('slug' => $slug));

    // ...

    // also change this line, since the $id variable is gone
    $deleteForm = $this->createDeleteForm($entity->getId());
    // ...
}
```

And with those 2 small changes, this page should work!

Updating the URL generation¶

Head over to the homepage to try it. Ah, a *huge* error:

> An exception has been thrown during the rendering of a template ("Some mandatory parameters are missing ("slug") to generate a URL for route "event_show".")

The `event_show` route now has a `slug` wildcard instead of `id`. So wherever we're generating a URL to this route, we need to change the wildcard we're passing to it.

I'll use the "git grep" command to figure out where we're using this route:

```
git grep event_show
```

Update each to pass in the `slug` instead of the `id`:

```php
// src/Yoda/EventBundle/Controller/EventController.php
// ..

public function createAction(Request $request)
{
    // ...

    return $this->redirect($this->generateUrl(
        'event_show', array('slug' => $entity->getSlug())
    ));
}
```

```twig
{# src/Yoda/EventBundle/Resources/views/Event/index.html.twig #}
{# ... #}

<a href="{{ path('event_show', {'slug': entity.slug}) }}">{{ entity.name }}</a>
```

```twig
{# src/Yoda/EventBundle/Resources/views/Event/edit.html.twig #}
{# ... #}

<a class="link" href="{{ path('event_show', {'slug': entity.slug}) }}">show event</a>
```

Refresh the homepage. Nice! When we click on an event, we have a beautiful URL.

Chapter 10: Adding createdAt and updatedAt Timestampable Fields

ADDING CREATEDAT AND UPDATEDAT TIMESTAMPABLE FIELDS¶

Let's do more magic! I always like to have `createdAt` and `updatedAt` fields on my database tables. A lot of times, this helps me debug any weird behavior I may see in the future.

The DoctrineExtensions library does this for us. It's called `timestampable`, enable it in `config.yml`:

```
# app/config/config.yml
# ...

stof_doctrine_extensions:
    orm:
        default:
            sluggable: true
            timestampable: true
```

Head to the timestampable section of the documentation to see how this works. We already have the `Gedmo` annotation, so just copy in the `created` and `updated` properties and rename them to `createdAt` and `updatedAt`, just because I like those names better:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @Gedmo\Timestampable(on="create")
 * @ORM\Column(type="datetime")
 */
private $createdAt;

/**
 * @Gedmo\Timestampable(on="update")
 * @ORM\Column(type="datetime")
 */
private $updatedAt;
```

And now we'll generate getter methods for these:

```
/**
 * @return \DateTime
 */
public function getCreatedAt()
{
    return $this->createdAt;
}

/**
 * @return \DateTime
 */
public function getUpdatedAt()
{
    return $this->updatedAt;
}
```

We can also add setter methods if we want, but we don't need them: the library will set these for us!

Next, update the database schema to add the two new fields and then reload the fixtures:

```
php app/console doctrine:schema:update --force
php app/console doctrine:fixtures:load
```

Query for the events again:

```
php app/console doctrine:query:sql "SELECT * FROM yoda_event"
```

Nice! Both the `createdAt` and `updatedAt` columns are properly set. To avoid sadness and regret add these fields to almost every table.

Chapter 11: Creating a Custom orderBy Query

CREATING A CUSTOM ORDERBY QUERY¶

Ok friends, the homepage lists every event in the order they were added to the database. We can do better! Head to `EventController` and replace the `findAll` method with a custom query that orders the events by the `time` property, so we can see the events that are coming up next first:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function indexAction()
{
    $em = $this->getDoctrine()->getManager();

    $entities = $em
        ->getRepository('EventBundle:Event')
        ->createQueryBuilder('e')
        ->addOrderBy('e.time', 'ASC')
        ->getQuery()
        ->execute()
    ;

    // ...
}
```

When we check the homepage, it looks about the same as before. Let's complicate things by only showing upcoming events:

```
$entities = $em
    ->getRepository('EventBundle:Event')
    ->createQueryBuilder('e')
    ->addOrderBy('e.time', 'ASC')
    ->andWhere('e.time > :now')
    ->setParameter('now', new \DateTime())
    ->getQuery()
    ->execute()
;
```

This uses the parameter syntax we saw before and uses a `\DateTime` object to only show events after right now.

To test this, edit one of the events and set its time to a date in the past. When we head back to the homepage, we see that the event is now missing from the list!

Moving Queries to the Repository¶

This is great, but what if we want to reuse this query somewhere else? Instead of keeping the query in the controller, create a new method called `getUpcomingEvents` inside `EventRepository` and move it there:

```
// src/Yoda/EventBundle/Entity/EventRepository.php
// ...

/**
 * @return Event[]
 */
public function getUpcomingEvents()
{
    return $this
        ->createQueryBuilder('e')
        ->addOrderBy('e.time', 'ASC')
        ->andWhere('e.time > :now')
        ->setParameter('now', new \DateTime())
        ->getQuery()
        ->execute()
    ;
}
```

Now that we're actually inside the repository, we just start by calling `createQueryBuilder()`. In the controller, continue to get the repository, but now just call `getUpcomingEvents` to use the method:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function indexAction()
{
    $em = $this->getDoctrine()->getManager();

    $entities = $em
        ->getRepository('EventBundle:Event')
        ->getUpcomingEvents()
    ;

    // ...
}
```

Note

The `$em->getRepository('EventBundle:Event')` returns our `EventRepository` object.

Whenever you need a custom query: create a new method in the right repository class and build it there. Don't create queries in your controller, seriously! We want your fellow programmers to be impressed when you show them your well-organized Jedi ways.

Chapter 12: ManyToMany Relationship

MANYTOMANY RELATIONSHIP¶

I want you to attend my event! So, you are going to need to be able to RSVP.

Adding a ManyToMany Relationship¶

First, think about how this would be stored in the database. One user should be able to attend many events, and one event will have many attendees. This is a classic `ManyToMany` relationship between the `Event` and `User` entities.

We already added a *ManyToOne relationship* earlier and adding a `ManyToMany` will be very similar.

To model this, create a new `attendees` property on `Event` that'll hold an array of Users that can't wait to go:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

protected $attendees;
```

Like with a `ManyToOne`, we just need an annotation that tells Doctrine what type of association this is and what entity it relates to:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToMany(targetEntity="Yoda\UserBundle\Entity\User")
 */
protected $attendees;
```

Whenever you have a relationship that holds multiple things, you need to add a `__construct` method and initialize it to an `ArrayCollection`:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

use Doctrine\Common\Collections\ArrayCollection;
// ...

public function __construct()
{
    $this->attendees = new ArrayCollection();
}
```

We saw this on the `User.events` property earlier when we added the *OneToMany association*.

Next, we'll add a `getter` method only - I'll explain why the `setter` isn't needed in a moment:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

public function getAttendees()
{
    return $this->attendees;
}
```

And that's it! Let's dump the schema update to see how this will change our database:

```
php app/console doctrine:schema:update --dump-sql
```

```
CREATE TABLE event_user (
    event_id INT NOT NULL,
    user_id INT NOT NULL,
    INDEX IDX_92589AE271F7E88B (event_id),
    INDEX IDX_92589AE2A76ED395 (user_id),
    PRIMARY KEY(event_id, user_id))
    DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
ALTER TABLE event_user
    ADD CONSTRAINT FK_92589AE271F7E88B FOREIGN KEY (event_id)
    REFERENCES yoda_event (id) ON DELETE CASCADE;
ALTER TABLE event_user
    ADD CONSTRAINT FK_92589AE2A76ED395 FOREIGN KEY (user_id)
    REFERENCES yoda_user (id) ON DELETE CASCADE;
```

Doctrine is smart enough to know that we need a new "join table" that has `event_id` and `user_id` properties. When we relate an `Event` to a `User`, it'll insert a new row in this table for us. Doctrine will handle all of those ugly details.

Re-run the command with `--force` to add the table:

```
php app/console doctrine:schema:update --force
```

The Optional JoinTable¶

With a `ManyToMany`, you can *optionally* add a `JoinTable` annotation. Add this only if you want to customize something about the join table. For example, you can control the onDelete behavior that happens if a User or Event is deleted:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\ManyToMany(targetEntity="Yoda\UserBundle\Entity\User")
 * @ORM\JoinTable(
 *     joinColumns={@ORM\JoinColumn(onDelete="CASCADE")},
 *     inverseJoinColumns={@ORM\JoinColumn(onDelete="CASCADE")}
 * )
 */
protected $attendees;
```

Run the `doctrine:schema:update` command again.

```
php app/console doctrine:schema:update --dump-sql
```

Actually, no changes are needed: Doctrine uses this onDelete behavior by default.

Chapter 13: Using the ManyToMany so Users can Attend an Event

USING THE MANYTOMANY SO USERS CAN ATTEND AN EVENT¶

Let's put our new relationship into action. Create two new routes next to our other event routes: one for attending an event and another for unattending:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
# ...

event_attend:
    pattern:  /{id}/attend
    defaults: { _controller: "EventBundle:Event:attend" }

event_unattend:
    pattern:  /{id}/unattend
    defaults: { _controller: "EventBundle:Event:unattend" }
```

Next, hop into the `EventController` and create the two corresponding action methods:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function attendAction($id)
{

}

public function unattendAction($id)
{

}
```

Start with `attendAction`. The logic here should feel familiar. First, query for an `Event` entity. Next, throw a `createNotFoundException` if no `Event` is found:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function attendAction($id)
{
    $em = $this->getDoctrine()->getManager();
    /** @var $event \Yoda\EventBundle\Entity\Event */
    $event = $em->getRepository('EventBundle:Event')->find($id);

    if (!$event) {
        throw $this->createNotFoundException('No event found for id '.$id);
    }

    // ... todo
}
```

All we need to do now is add the current `User` object as an attendee on this `Event`. Remember that the `attendees` property is actually an `ArrayCollection` object. Use its `add` method then save the `Event`. Finally, redirect when you're finished:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function attendAction($id)
{
    $em = $this->getDoctrine()->getManager();
    /** @var $event \Yoda\EventBundle\Entity\Event */
    $event = $em->getRepository('EventBundle:Event')->find($id);

    if (!$event) {
        throw $this->createNotFoundException('No event found for id '.$id);
    }

    $event->getAttendees()->add($this->getUser());

    $em->persist($event);
    $em->flush();

    $url = $this->generateUrl('event_show', array(
        'slug' => $event->getSlug(),
    ));

    return $this->redirect($url);
}
```

Notice that we just added an attendee without needing a `setAttendees` method on `Event`. This works because `attendees` is an object, so we can just call `getAttendees` and then modify it.

Printing Attendees in Twig¶

Before we try this out, let's update the event show page. Use the `length` filter to count the number of attendees, to make sure we make enough guacamole:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

<dt>who:</dt>
<dd>
    {{ entity.attendees|length }} attending!

    <ul class="users">
        <li>nobody yet!</li>
    </ul>
</dd>
```

We can even loop over the event's attendees and print each of them out. Print a nice message when nobody's attending, using Twig's really nice for-else functionality:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

<dt>who:</dt>
<dd>
    {{ entity.attendees|length }} attending!

    <ul class="users">
        {% for attendee in entity.attendees %}
            <li>{{ attendee }}</li>
        {% else %}
            <li>nobody yet!</li>
        {% endfor %}
    </ul>
</dd>
```

Now help me add a link to the new `event_attend` route if the user is logged in:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

<dt>who:</dt>
<dd>
    {# ... #}

    <a href="{{ path('event_attend', {'id': entity.id}) }}" class="btn btn-success btn-xs">
        I totally want to go!
    </a>
</dd>
```

Testing out the Relationship¶

Head over to an event in your browser. It says 0 attending. Now click the new link. After the redirect, we see 1 attending, but we also see a huge error:

> Catchable Fatal Error: Object of class YodaUserBundleEntityUser could not be converted to string

The fact that we show 1 attending means that the database relationship was stored correctly. We can prove it by querying for the join table:

```
php app/console doctrine:query:sql "SELECT * FROM event_user"
```

Yep, we see one row that links our user to this event.

Adding a __toString to User¶

So what's the error? Look closely: PHP is trying to convert our `User` object into a string. This is happening because we're looping over `event.attendees`, which gives us User objects that we're printing:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}

{% for attendee in entity.attendees %}
    <li>{{ attendee }}</li>
{% else %}
    <li>nobody yet!</li>
{% endfor %}
```

We have two options to fix this. First, we *could* just print out a specific property on the `User`:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}

{% for attendee in entity.attendees %}
    <li>{{ attendee.username }}</li>
{% else %}
    <li>nobody yet!</li>
{% endfor %}
```

But if you *do* just want to print the object, you can add a `__toString` method to the `User` class:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

public function __toString()
{
    return (string) $this->getUsername();
}
```

Refresh now. Sweet, no errors!

Let's also take a second and fill in the # of attendees on the *index* page:

```
{# src/Yoda/EventBundle/Resources/views/Event/index.html.twig #}
{# ... #}

{% for entity in entities %}
    {# ... #}

    <dt>who:</dt>
    <dd>{{ entity.attendees|length }} attending!</dd>

    {# ... #}
{% endfor %}
```

Chapter 14: More with ManyToMany: Avoiding Duplicates

MORE WITH MANYTOMANY: AVOIDING DUPLICATES¶

Now click the attend link again. Ah, an error!

> SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate entry '4-4' for key 'PRIMARY'

Our User is once again added as an attendee to the Event. And when Doctrine saves, it tries to add a second row to the join table. Not cool!

Adding the hasAttendee Method¶

To fix this, create a new method in `Event` called `hasAttendee`. This will return true or false depending on whether or not a given user is attending this event:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @param \Yoda\UserBundle\Entity\User $user
 * @return bool
 */
public function hasAttendee(User $user)
{
    return $this->getAttendees()->contains($user);
}
```

Avoiding Duplicates¶

Find `attendAction` in `EventController`. We can use the new `hasAttendee` method to avoid adding duplicate Users:

```
// src/Yoda/EventBundle/Controller/EventController.php

public function attendAction($id)
{
    // ...

    if (!$event->hasAttendee($this->getUser())) {
        $event->getAttendees()->add($this->getUser());
    }

    // ...
}
```

Try it out! Go crazy, click the attend link as many times as you want: you're only added the first time.

Adding Unattend Logic¶

Let's fill in the logic in `unattendAction`. Actually, we can just copy `attendAction` and *remove* the current user from the attendee list by using the `removeElement` method:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function unattendAction($id)
{
    $em = $this->getDoctrine()->getManager();
    /** @var $event \Yoda\EventBundle\Entity\Event */
    $event = $em->getRepository('EventBundle:Event')->find($id);

    if (!$event) {
        throw $this->createNotFoundException('No event found for id '.$id);
    }

    if ($event->hasAttendee($this->getUser())) {
        $event->getAttendees()->removeElement($this->getUser());
    }

    $em->persist($event);
    $em->flush();

    $url = $this->generateUrl('event_show', array(
        'slug' => $event->getSlug(),
    ));

    return $this->redirect($url);
}
```

In our show template, let's show only the "attend" or "unattend" link based on whether we're attending the event or not. That's easy with the `hasAttendee` method:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

<dt>who:</dt>
<dd>
    {# ... #}

        {% if entity.hasAttendee(app.user) %}
            <a href="{{ path('event_unattend', {'id': entity.id}) }}" class="btn btn-warning btn-xs">
                Oh no! I can't go anymore!
            </a>
        {% else %}
            <a href="{{ path('event_attend', {'id': entity.id}) }}" class="btn btn-success btn-xs">
                I totally want to go!
            </a>
        {% endif %}
</dd>
```

When we refresh, the unattend button is showing. Click it and then click the attend button again. This bake sale is going to be off the hook!

What's really going on in the Base Controller¶

Quickly, look back at the `redirect` and `generateUrl` methods we're using in our controller. Let's see what these really do by opening up `Symfony's base controller` class:

```php
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php
// ...

public function generateUrl($route, $parameters = array(), $absolute = false)
{
    return $this->container->get('router')->generate($route, $parameters, $absolute);
}

public function redirect($url, $status = 302)
{
    return new RedirectResponse($url, $status);
}
```

Like we've seen over and over again, `generateUrl` is just a shortcut to grab a service from the container and call a method on it. The `redirect` method is even simpler: it returns a special type of `Response` object that's used when redirecting users.

The point is this: Symfony is actually pretty simple under the surface. Your job in every controller is to return a `Response` object. The container gives you access to all types of powerful objects to make that job easier.

Chapter 15: JSON up in your Response

JSON UP IN YOUR RESPONSE¶

Yea, we can RSVP for an event. But it's not super-impressive yet. You and I both know that a little AJAX could spice things up.

Creating JSON-returning Actions for AJAX¶

Our attend and unattend endpoints aren't really ready for AJAX yet. They both return a redirect response, which really only makes sense when you want the browser to do full page refreshes.

So why not return something different, like a JSON response? JSON is great because it's easy to create in PHP and easy for JavaScript to understand. And actually, could we make the endpoints return both? Why not!

Start by adding a `format` wildcard to both of the routes. Give it a default value of `html`:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
# ...

event_attend:
    pattern:  /{id}/attend.{format}
    defaults: { _controller: "EventBundle:Event:attend", format: html }

event_unattend:
    pattern:  /{id}/unattend.{format}
    defaults: { _controller: "EventBundle:Event:unattend", format: html }
```

As soon as we give a wildcard a default value, it makes it optional. For us, it means that we can now go to `/5/attend.json`, but `/5/attend` still works too. So if the `format` part is missing, the route still matches.

In a truly RESTful API, it's more "correct" to read the `Accept` header instead of putting the format in the URL like we're doing here. If you're interested in that, check out our REST Series, it'll blow your mind.

Routing Wildcard requirements¶

I don't really feel like also making the endpoints return XML, so let's add a `requirements` key to the route:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
# ...

event_attend:
    pattern:  /{id}/attend.{format}
    defaults: { _controller: "EventBundle:Event:attend", format: html }
    requirements:
        format: json

event_unattend:
    pattern:  /{id}/unattend.{format}
    defaults: { _controller: "EventBundle:Event:unattend", format: html }
    requirements:
        format: json
```

Now try going to the URL with `.xml` in the end. The route doesn't match! Requirements are little regular expressions that you can use to restrict any wildcard.

Returning a JSON Response from a Controller¶

With this new wildcard in our route, we can now use it to return JSON *or* a redirect response.

You know what the next step is: give `attendAction` a `$format` argument:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function attendAction($id, $format)
{
    // ...
}
```

If it's equal to `json`, we can return a JSON string instead of a redirect:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

use Symfony\Component\HttpFoundation\Response;

public function attendAction($id, $format)
{
    // ...
    $em->flush();

    if ($format == 'json') {
        $data = array(
            'attending' => true
        );

        $response = new Response(json_encode($data));

        return $response;
    }

    return $this->redirect($this->generateUrl('event_show', array(
        'slug' => $event->getSlug()
    )));
}
```

How? Just create an array and then convert it to JSON with `json_encode`. And do you remember the cardinal rule of controllers? A controller *always* returns a Symfony Response object. So just create a new `Response` object and set the JSON as its body. It's that simple, stop over-complicating it!

Test it out by copying the link and adding `.json` to the end. Hello, beautiful JSON!

Tip

The JSON is pretty in my browser because of the JSONView Chrome extension.

Chapter 16: Come on, Set the Content-Type Header!

COME ON, SET THE CONTENT-TYPE HEADER!¶

If you go to the network tab of your brower's tools and refresh, you'll find an ugly surprise. Our response has a `text/html` `Content-Type` ! Silly browser!

Ok, this is our fault. Every response has a `Content-Type` header and its job is to tell the client if the page is `text/html` , `application/json` , or `text/turtle` . Yea, that's a real format. It's actually XML, so not as cute as the name sounds.

Anyways, it's *our* job to set this header, which defaults to `text/html` in Symfony. Use the `headers` property on the `$response` to set it:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function attendAction($id, $format)
{
    // ...

    if ($format == 'json') {
        // ...

        $response = new Response(json_encode($data));
        $response->headers->set('Content-Type', 'application/json');

        return $response;
    }

    // ...
}
```

Alright! Refresh again. Mmm, a beautiful `application/json` `Content-Type` .

The JsonResponse Class¶

Ok, so there's an even *lazier* way to do this. So throw on your sweat pants, grab that bag of chips and let's get *lazy*. Instead of `Response` , use a class called `JsonResponse` and pass it the array directly. Oh, and get rid of the `Content-Type` header while you're in there:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

use Symfony\Component\HttpFoundation\JsonResponse;

public function attendAction($id, $format)
{
    // ...

    if ($format == 'json') {
        $data = array(
            'attending' => true
        );

        $response = new JsonResponse($data);

        return $response;
    }

    // ...
}
```

Refresh again. Yea, we still see JSON *and* the `Content-Type` header is still `application/json`. `JsonResponse` is just a sub-class of `Response`, but it removes a few steps for us, and I like that.

Finishing up the Controller¶

Time to stop playing and finish `unattendAction`. Just copy the logic from `attendAction`, change the value to `false`, and don't forget the `$format` argument:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function unattendAction($id, $format)
{
    // ...
    $em->flush();

    if ($format == 'json') {
        $data = array(
            'attending' => false
        );

        $response = new JsonResponse($data);

        return $response;
    }

    $url = $this->generateUrl('event_show', array(
        'slug' => $event->getSlug()
    ));

    return $this->redirect($url);
}
```

When we try it manually, it seems to work!

Removing Duplication¶

Looking at these 2 methods, do you see any duplication? Um, yea, just about every line is duplicated. We can fix at least some of this by creating a new private method called `createAttendingResponse` with `$event` and `$format` arguments.

Copy in the logic that figures out which response to return:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

/**
 * @param Event $event
 * @param string $format
 * @return \Symfony\Component\HttpFoundation\Response
 */
private function createAttendingResponse(Event $event, $format)
{
    if ($format == 'json') {
        $data = array(
            'attending' => $event->hasAttendee($this->getUser())
        );

        $response = new JsonResponse($data);

        return $response;
    }

    $url = $this->generateUrl('event_show', array(
        'slug' => $event->getSlug()
    ));

    return $this->redirect($url);
}
```

For the `attending` value, why not just use our `hasAttendee` method to figure this out?

Sweet, let's do my favorite thing – delete some code! Call the new method in `attendAction` and `unattendAction` and return its value.

We can use this function to easily generate the JSON response for both controllers:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function attendAction($id, $format)
{
    // ...

    return $this->createAttendingResponse($event, $format);
}

public function unattendAction($id, $format)
{
    // ...

    return $this->createAttendingResponse($event, $format);
}
```

Try it out! Isn't it nice when things *don't* break?

Chapter 17: Adding the AJAX Touch: JavaScript

ADDING THE AJAX TOUCH: JAVASCRIPT¶

Stop. We haven't touched JavaScript yet. But, because the attend and unattend endpoints can return JSON, our app is fully ready for some AJAX. The attend/unattend button will be a lot cooler with it anyways, so let's add some JavaScript.

Click Event to Send AJAX¶

I'll give both links a `js-attend-toggle` class that we can look for in jQuery:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

    {% if entity.hasAttendee(app.user) %}
        <a href="{{ path('event_unattend', {'id': entity.id}) }}"
            class="btn btn-warning btn-xs js-attend-toggle">

            Oh no! I can't go anymore!
        </a>
    {% else %}
        <a href="{{ path('event_attend', {'id': entity.id}) }}"
            class="btn btn-success btn-xs js-attend-toggle">

            I totally want to go!
        </a>
    {% endif %}
```

Adding the JavaScript¶

Wait! We can't write jQuery without, ya know, including jQuery. So open up the base template and add it inside the `javascripts` block. I'm just going to use a CDN:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% block javascripts %}
    <script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
{% endblock %}
```

To add JavaScript on just this page, we can override this block and call the `parent()` function. I'll paste in some jQuery magic that makes an AJAX call when the links are clicked. You can get this magic from the `attend-javascript.js` file in the code download:

```
{# src/Yoda/EventBundle/Resources/views/Event/show.html.twig #}
{# ... #}

{% block javascripts %}
    {{ parent() }}

    <script>
        $(document).ready(function() {
            $('.js-attend-toggle').on('click', function(e) {
                // prevents the browser from "following" the link
                e.preventDefault();

                var $anchor = $(this);
                var url = $(this).attr('href')+'.json';

                $.post(url, null, function(data) {
                    if (data.attending) {
                        var message = 'See you there!';
                    } else {
                        var message = 'We\'ll miss you!';
                    }

                    $anchor.after('<span class="label label-default">&#10004; '+message+'</span>');
                    $anchor.hide();
                });
            });
        });
    </script>
{% endblock %}
```

I know. In a perfect world, this should live in an external JavaScript file. I'll leave that to you.

Let's try our new AJAX magic! Ooh, fancy. The link disappears and we get a cute message.

The code is simple enough: we listen on a click of either link, send an AJAX request, then hide the link and show a message. To get the URL, I'm using the href then adding `.json` to the end of it. That's actually kinda hacky. There's a sweet bundle called FOSJsRoutingBundle that can do this much better. It let's you actually generate Symfony routes right in JavaScript.

It's easy to use, so include it in your projects!

Chapter 18: Customizing Error Pages and How Errors are Handled

CUSTOMIZING ERROR PAGES AND HOW ERRORS ARE HANDLED¶

Sometimes things fall apart. And when they do, we show our users an error page. Hopefully, a hilarious error page.

Right now, our 404 page isn't very hilarous, except for the little Pacman ghost that's screaming "Exception detected". He's adorable.

We see this big descriptive error page because we're in the `dev` environment, and Symfony wants to help us fix our mistake. In real life, also known as the `prod` environment, it's different.

The Real Life: prod Environment¶

To see our app in its "real life" form, put an `app.php` after `localhost`:

> http://localhost:9000/app.php

We talked about environments and this `app.php` stuff in episode 1. If you don't remember it, go back and check it out!

The page *might* work or it might be broken. That's because we always need to clear our Symfony cache when going into the `prod` environment:

```
php app/console cache:clear --env=prod
```

Ok, now the site works. Invent a URL to see the 404 page. Ah gross! This error page isn't hilarous at all! So where is the content for this page actually coming from and how can we make a better experience for our users?

Overriding the Error Template Content¶

To find out, let's just search the project! In PHPStorm, I can navigate to `vendor/symfony/symfony`, right click, then select "Find in Path". Let's look for the "An Error Occurred" text.

Ah hah! It points us straight to a file in the core Twig bundle called `error.html.twig`. Let's open that up!

Tip

The location of the file is:

> vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/Resources/views/Exception/error.html.twig

Cool, so how can we replace this with a template that has unicorns, or pirates or anything better than this?

There's actually a *really* neat trick that let's you override *any* template from *any* bundle. All we need to do is create a template with the same name as this in *just* the right location.

This template lives in `TwigBundle` and in an `Exception` directory. Create an `app/Resources/TwigBundle/views/Exception/error.html.twig` file. Notice how similar the paths are - it's the magic way to override *any* template from *any* bundle.

Tip

`app/Resources/AnyBundle/views/SomeDir/myTemplate.html.twig` will always override `@AnyBundle/Resources/views/SomeDir/myTemplate.html.twig`

Now just extend the base template and put something awesome inside. I'm going to abuse my `login.css` file to get this to look ok. I know, I really need to clean up my CSS:

```
{# app/Resources/TwigBundle/views/Exception/error.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}
    <link rel="stylesheet" href="{{ asset('bundles/user/css/login.css') }}" />
{% endblock %}

{% block body %}
    <section class="login">
        <article>
            <h1>Ah crap!</h1>

            <div>These are not the droids you're looking for...</div>
        </article>
    </section>
{% endblock %}
```

Refresh! Hey, don't act so surprised to see the same ugly template. We're in the `prod` environment, we need to clear our cache after every change:

```
php app/console cache:clear --env=prod
```

Refresh again. It's beautiful. The pain with customizing error templates is that you need to be in the `prod` environment to see them. And that means you need to remember to clear cache after every change.

Customizing Error Pages by Type (Status Code)¶

But we have a problem: this template is used for *all* errors: 404 errors, 500 errors and even the dreaded 418 error!

I think we should at least have one template for 404 errors and another for everything else. Copy the existing template and paste it into a new file called `error404.html.twig`. That's the trick, and this works for customizing the error page of any HTTP status code.

We should keep the generic error template, but let's give it a different message:

```
{# app/Resources/TwigBundle/views/Exception/error.html.twig #}

{# ... #}
<h1>Ah crap!</h1>

<div>The servers are on fire! Grab a bucket! Send halp!</div>
```

To see the 404 template, clear your cache and refresh again on an imaginary URL. To see the other template, temporarily throw an exception in `EventController::indexAction` to cause a 500 error:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function indexAction()
{
    throw new \Exception('Ahhhhahahhhah');
    // ...
}
```

Head to the homepage - but with the `app.php` still in the URL. You should see that the servers are in fact on fire, which I guess is cool. Remove this exception before moving on.

Going Deeper with Exception Handling¶

Behind the scenes, Symfony dispatches an event whenever an exception happens. We haven't talked about events yet, but this basically means that if you want, you can be nofitied whenever an exception is thrown anywhere in your code. Why would you do this? You might want to do some extra logging or even completely replace which template is rendered when an error happens.

We won't cover event listeners in this screencast, but there's a cookbook called How to Create an Event Listener that covers it.

Normally, when there's an exception, Symfony calls an internal controller that renders the error template. This class lives in TwigBundle and is called `ExceptionController`. Let's open it up!

> The class lives at: vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/Controller/ExceptionController.php

The guts of this class aren't too important, but you *can* see it trying to figure out which template to render in `findTemplate`. You can even see it looking for the status-code version of the template, like `error404.html.twig`:

```
// vendor/symfony/symfony/src/Symfony/Bundle/TwigBundle/Controller/ExceptionController.php
// ...

$template = new TemplateReference('TwigBundle', 'Exception', $name.$code, $format, 'twig');
if ($this->templateExists($template)) {
    return $template;
}
```

I'm making you stare at this class because, if you want, you can actually override this entire controller. If you do that, then *your* controller function will be called whenever there's an error and *you* can render whatever page you want. That process is a bit more involved, but use it if you need to go even further.

Chapter 19: Render another Controller in Twig

RENDER ANOTHER CONTROLLER IN TWIG¶

When a user sees our 404 page, I'd *love* it if we could show them a list of upcoming events. Hmm, but that's not possible. Normally, I'd query for some events and then pass them into my template. But we don't have access to Symfony's core controller that's rendering `error404.html.twig`.

Whenever you're in a template and don't have access to something you need, there's a sure-fire solution: use the Twig `render` function. This lets you call any controller function you want and prints the results.

Create an Embedded Controller¶

Start by adding a new controller function that queries for upcoming events, and renders a template. So far, this feels like any other controller, except it doesn't have a route:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function _upcomingEventsAction()
{
    $em = $this->getDoctrine()->getManager();

    $events = $em->getRepository('EventBundle:Event')
        ->getUpcomingEvents()
    ;

    return $this->render('EventBundle:Event:_upcomingEvents.html.twig', array(
        'events' => $events,
    ));
}
```

Create the template and grab the event-rendering code from `index.html.twig`. But hey, *don't* extend the base layout. This controller is meant to just render "part" of a page, not the entire HTML body. I also need to rename `entities` to `events`, since that's how I called the variable in the controller:

```
{# src/Yoda/EventBundle/Resources/views/Event/_upcomingEvents.html.twig #}
{% for event in events %}
  <article>
    <header class="map-container">
      <img src="http://maps.googleapis.com/maps/api/staticmap?center={{ event.location | url_encode }}&markers=color:r
    </header>
    <section>
      <h3>
        <a href="{{ path('event_show', {'slug': event.slug}) }}">{{ event.name }}</a>
      </h3>

      <dl>
        <dt>where:</dt>
        <dd>{{ event.location }}</dd>

        <dt>when:</dt>
        <dd>{{ event.time | date('g:ia / l M j, Y') }}</dd>

        <dt>who:</dt>
        <dd>Todo # of people</dd>
      </dl>
    </section>
  </article>
{% endfor %}
```

The new controller prints *just* a list of events, without a layout. We didn't give it a route, but we don't need to: we're going to call it straight from Twig.

Oh, and what's up with the underscore in front of the name? That's just a standard I follow for controllers that render partial pages.

Getting render-happy in Twig¶

Ok, *now* I'll show you the power behind this render weapon. Remove the query in `indexAction` and pass nothing into the template:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

/**
 * @Template()
 * @Route("/", name="event")
 */
public function indexAction()
{
  return array();
}
```

Next, remove the big `entities` for loop that we just copied from `index.html.twig` and replace it with the `render` function:

```
{% extends 'EventBundle::layout.html.twig' %}

{% block body %}
  <section class="events">
    {# same <header> stuff as before #}
    {# ... #}

    {{ render(controller('EventBundle:Event:_upcomingEvents')) }}
  </section>
{% endblock %}
```

Try out the homepage in the `dev` environment. Hey, it looks just like before! `render` calls our controller, we build a partial HTML page, and then it gets printed. This handy function is great for re-using page chunks and is also key to using Symfony's Caching.

Tip

If you just want to re-use a Twig template, use the include function.

Using render in the Error Template¶

Our goal was to list upcoming events on the 404 page. Well, that's pretty easy now:

```
{# app/Resources/TwigBundle/views/Exception/error404.html.twig #}
{# ... #}

{% block body %}
    {# existing <section> ... #}

    <section class="events">
        {{ render(controller('EventBundle:Event:_upcomingEvents')) }}
    </section>
{% endblock %}
```

Move to an imaginary page in your `prod` environment. In other words, put the `app.php` back in the URL:

http://localhost:9000/app.php/foo

Ah, but don't forget to clear your cache!

```
php app/console cache:clear --env=prod
```

Controller Arguments¶

Great! Now what if we wanted to show a different number of upcoming events on the homepage versus the error page? No problem: `render` let's us pass arguments to the controller function. Pass a `max` argument of `1` from the error template:

```
{# app/Resources/TwigBundle/views/Exception/error404.html.twig #}
{# ... #}

<section class="events">
    {{ render(controller('EventBundle:Event:_upcomingEvents', {
        'max': 1
    })) }}
</section>
```

Next, add a `$max` argument to `_upcomingEventsAction` and give it a default value so that we don't *have* to pass it in. Send this variable into the `getUpcomingEvents()` function:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

public function _upcomingEventsAction($max = null)
{
    $em = $this->getDoctrine()->getManager();

    $events = $em->getRepository('EventBundle:Event')
        ->getUpcomingEvents($max)
    ;

    return $this->render('EventBundle:Event:_upcomingEvents.html.twig', array(
        'events' => $events,
    ));
}
```

In `EventRepository`, give the function a `$max` argument. Instead of returning immediately, set the query builder to a variable and then return it later. If `$max` is set, limit the number of results that will be returned:

```
// src/Yoda/EventBundle/Entity/EventRepository.php
// ...

public function getUpcomingEvents($max = null)
{
    $qb = $this
        ->createQueryBuilder('e')
        ->addOrderBy('e.time', 'ASC')
        ->andWhere('e.time > :now')
        ->setParameter('now', new \DateTime());

    if ($max) {
        $qb->setMaxResults($max);
    }

    return $qb
        ->getQuery()
        ->execute()
    ;
}
```

Clear your cache and then try it out. Hey, only 1 event! Not only can `render` call a controller, but we can control its arguments. Now you're unstoppable.

Chapter 20: Creating a Pretty CSV Download

CREATING A PRETTY CSV DOWNLOAD¶

Buzzword time! Services! Dependency Injection! Dependency Injection Container!

Hold on, because we're about to discover what these terms mean, how *core* they are to Symfony, and just how simple these things really are.

Pretend that someone needs to be able to download a CSV of all of the events that have been updated during the last 24 hours. Let's create a whole new controller class for this called `ReportController`, since `EventController` is getting a bit big:

```
// src/Yoda/EventBundle/Controller/ReportController.php
namespace Yoda\EventBundle\Controller;

class ReportController extends Controller
{

}
```

Let's create an `updatedEventsAction` and use those handy annotation routes. And of course don't forget to copy in the `Route use` statement for the annotation:

```
/**
 * @Route("/events/report/recentlyUpdated.csv")
 */
public function updatedEventsAction()
{

}
```

Try the URL in your browser. If you see the "The controller must return a response" error like I do, then we're good! This is proof that our controller is being executed.

Creating a CSV Response¶

First we need a custom query to find recently updated events. Should we just put this in our controller? I *hope* you're screaming no. The force is strong enough in us to now put these directly in our repository class. Create a new `getRecentlyUpdatedEvents` method in `EventRepository` and build a query that *only* returns events updated within the last 24 hours:

```
// src/Yoda/EventBundle/Entity/EventRepository.php
// ...

public function getRecentlyUpdatedEvents()
{
    return $this->createQueryBuilder('e')
        ->andWhere('e.updatedAt > :since')
        ->setParameter('since', new \DateTime('24 hours ago'))
        ->getQuery()
        ->execute()
    ;
}
```

Let's call this in the controller. This should be getting boring because, we always query the same way: get the entity manager, get the repository, then call a method on it:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

public function updatedEventsAction()
{
    $em = $this->getDoctrine()->getManager();

    $events = $em->getRepository('EventBundle:Event')
        ->getRecentlyUpdatedEvents();

    // ...
}
```

Now we need to turn these `Event` objects into a CSV. I'll write some manual code for this. Yes, there *are* better ways to create CSV's, but trust me for a second. This code will help us show off one of Symfony's most powerful features:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

public function updatedEventsAction()
{
    $em = $this->getDoctrine()->getManager();

    $events = $em->getRepository('EventBundle:Event')
        ->getRecentlyUpdatedEvents();

    $rows = array();
    foreach ($events as $event) {
        $data = array($event->getId(), $event->getName(), $event->getTime()->format('Y-m-d H:i:s'));

        $rows[] = implode(',', $data);
    }

    $content = implode("\n", $rows);

    // ...
}
```

So what does a controller *always* return? A Response object of course! Let's just create one manually and pass the csv `$content` to it:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...
use Symfony\Component\HttpFoundation\Response;

public function updatedEventsAction()
{
    // ...

    $content = implode("\n", $rows);
    $response = new Response($content);

    return $response;
}
```

Refresh! Gosh, that's the prettiest CSV I've seen all day. Ah, but if I check the network tab in my browser, the response is `text/html`. I forgot to set that pesky `Content-Type` header. Let's fix that:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

public function updatedEventsAction()
{
    // ...

    $content = implode("\n", $rows);
    $response = new Response($content);
    $response->headers->set('Content-Type', 'text/csv');

    return $response;
}
```

This time Chrome sees that it's a CSV and downloads it for me. There's nothing new so far, but we're writing great code.

Chapter 21: Your Very First Service

YOUR VERY FIRST SERVICE¶

Create a new `Reporting` directory in the bundle and a new `EventReportManager` class inside of it:

```
// src/Yoda/EventBundle/Reporting/EventReportManager.php
namespace Yoda\EventBundle\Reporting;

class EventReportManager
{
}
```

Like any other class, give it the right namespace.

But, unlike entities, forms and controllers, this class is special because it has *absolutely* nothing to do with Symfony. It's just a "plain-old-PHP-object" that we'll use to help organize our own code.

Create a `getRecentlyUpdatedReport` method to the class and paste the logic from our controller that creates the CSV text:

```
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

class EventReportManager
{
    public function getRecentlyUpdatedReport()
    {
        $em = $this->getDoctrine()->getManager();

        $events = $em->getRepository('EventBundle:Event')
            ->getRecentlyUpdatedEvents();

        $rows = array();
        foreach ($events as $event) {
            $data = array($event->getId(), $event->getName(), $event->getTime()->format('Y-m-d H:i:s'));

            $rows[] = implode(',', $data);
        }

        return implode("\n", $rows);
    }
}
```

To use it in `ReportController`, create a new instance of `EventReportManager` and call `getRecentlyUpdatedReport` on it:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

use Yoda\EventBundle\Reporting\EventReportManager;

public function updatedEventsAction()
{
    $eventReportManager = new EventReportManager();
    $content = $eventReportManager->getRecentlyUpdatedReport();

    $response = new Response($content);
    $response->headers->set('Content-Type', 'text/csv');

    return $response;
}
```

And hey! Don't forget the `use` statement when referencing the class.

So why am I making you do this? Remember how we put our queries in repository classes? That's cool because it keeps things organized and we can also re-use those queries.

We're doing the same exact thing, but for reporting code instead of queries. Inside `EventReportManager`, the reporting code is reusable and organized in one spot.

DEPENDENCYINJECTION TO THE RESCUE!¶

But don't get too excited, I broke our app. Sorry. Refresh to see the error:

> Call to undefined method YodaEventBundleReportingEventReportManager::getDoctrine()

We're calling `$this->getDoctrine()`. That function lives in Symfony's base Controller. But in `EventReportManager`, we don't extend anything and we don't magically have access to this Doctrine object.

The code inside `EventReportManager` is *dependent* on this "doctrine" object. Well, more specifically, it's dependent on Doctrine's entity manager.

The fix for our puzzle is to "inject the dependency", or to use "dependency injection". That's a very scary term for a really simple idea.

First, add a constructor method with a single `$em` argument. Set that on a new `$em` class property:

```
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

class EventReportManager
{
    private $em;

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

    // ...
}
```

This will be the entity manager object. Inside `getRecentlyUpdatedReport`, use the new `$em` property and remove the non-existent `getDoctrine` call:

```
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

private $em;
// ...

public function getRecentlyUpdatedReport()
{
    $events = $this->em->getRepository('EventBundle:Event')
        ->getRecentlyUpdatedEvents();

    // ...
}
```

Back in `ReportController`, get the entity manager like we always do and pass it as the first argument when creating a new `EventReportManager`:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

use Yoda\EventBundle\Reporting\EventReportManager;

public function updatedEventsAction()
{
    $em = $this->getDoctrine()->getManager();
    $eventReportManager = new EventReportManager($em);
    $content = $eventReportManager->getRecentlyUpdatedReport();

    // ...
}
```

Refresh! Yes! The CSV has downloaded!

You deserve some congrats. You've just done "dependency injection". It's not some new programming practice or magic trick, it's just the idea of passing dependencies into objects that need them. For us, `EventReportManager` needs the entity manager object. So when we create the manager, we just "inject" it by passing it to the constructor. Now that the manager has everything it needs, it can get its work done.

Tip

To learn more, check out our free tutorial that's all about the great topic of Dependency Injection.

SO WHAT'S A SERVICE?¶

And you know what else? We also just created our first "service". Yes, we're hitting multiple buzzwords at once!

A "service" is a term that basically refers to any object that does some work for us. `EventReportManager` generates a CSV, so it's a "service".

So what's an object that's *not* a service? How about an entity. They don't really *do* anything, they just hold data. If you code well, you'll notice that every class fits into one of these categories. A class either does work but doesn't hold much data, like a service, or it holds data but doesn't do much, like an entity.

Another common property of a "service" class is that you only ever need one instance at a time. If we needed to generate 2 CSV reports, it wouldn't really make sense to instantiate 2 `EventReportManager` objects when we can just re-use the same one twice. "Services" are the machines of your app: each does its own "work", like creating reports, sending emails, or anything else you can dream up.

Chapter 22: Symfony Overlord: The Service Container

SYMFONY OVERLORD: THE SERVICE CONTAINER¶

One more buzzword: the service container, or dependency injection container. The service container is the benevolent overlord that's behind everything. He doesn't do any work, but he controls all the little peons, or services.

Accessing Existing Services¶

The container is just a simple object that holds *all* of the services in your project, including Symfony's core objects. Run the `container:debug` console task to get a list of these:

```
php app/console container:debug
```

The list is tiny, only about 200 or so. With such a tiny list, it's easy to spot the entity manager service: `doctrine.orm.entity_manager`. This is the "name" of the service and we use it to get this object out of the service container.

We've been getting the entity manager by using a helper function in the controller. But since we know its service name, we can get it directly:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

public function updatedEventsAction()
{
    $em = $this->container->get('doctrine.orm.entity_manager');
    $eventReportManager = new EventReportManager($em);
    $content = $eventReportManager->getRecentlyUpdatedReport();

    // ...
}
```

Refresh! The download still works: this is just a more direct way to access the same object. But stop! This is *hugely* powerful! Symfony's container holds over 200 services, and you can get *any* of these in a controller and use them. It's like someone just gave you 200 new power tools! You may not know how to use them yet, but you're about to look like Edward Scissorhands!

Adding a Service¶

I want to go further by adding our own service to the container.

Find and open a `services.yml` file that was generated automatically in `EventBundle`. When you add a new service, you're "teaching" the container how to instantiate it. First, it needs to know what the class name is:

```
# src/Yoda/EventBundle/Resources/config/services.yml
services:
    event_report_manager:
        class: Yoda\EventBundle\Reporting\EventReportManager
        arguments: []
```

The `event_report_manager` is the internal name of the service and can be anything.

The `arguments` key tells the container exactly what to pass to the constructor when it creates a new instance of our service. For example, if the first `__construct` argument to `EventReportManager` were a string, we could just type that value here:

```
# src/Yoda/EventBundle/Resources/config/services.yml
services:
    event_report_manager:
        class: Yoda\EventBundle\Reporting\EventReportManager
        arguments: [foo]
```

But instead of a string, the first argument to `EventReportManager` is the entity manager *service* object. To pass in a service, just put its name here and prefix it with the magic `@` symbol:

```
# src/Yoda/EventBundle/Resources/config/services.yml
services:
    event_report_manager:
        class: Yoda\EventBundle\Reporting\EventReportManager
        arguments: ["@doctrine.orm.entity_manager"]
```

The `@` symbol tells the container that `doctrine.orm.entity_manager` isn't a string: it's another object inside the container. When the container creates a new instance of `EventReportManager`, it passes the entity manager to it.

Re-run the `container:debug` console command:

```
php app/console container:debug
```

Ooo la la! Our new service is in the container.

Using the New Service¶

Get this new service in our controller. You already know how to get objects out of the container - we just did it a minute ago with the entity manager. It's exactly the same with *our* service.

In `ReportController`, remove the new call of the `EventReportManager` and replace it with a call to the `container` object:

```
// src/Yoda/EventBundle/Controller/ReportController.php
// ...

public function updatedEventsAction()
{
    $eventReportManager = $this->container->get('event_report_manager');
    $content = $eventReportManager->getRecentlyUpdatedReport();

    // ...
}
```

Refresh! Bam, the CSV still downloads. Internally, Symfony creates a new instance of `EventReportManager` and returns it. If we asked for the service a second time, the container would just give us the same instance as before, instead of creating a new one. That's nice for performance.

Back up and look at what we've accomplished. By creating `EventReportManager` and moving logic there, we made some of our code more organized and reusable. By going a step further and registering a service, we made it *even* easier to get and use this object. The services on the container are your application's *tools*, and you'll add more and more.

Hey Look at this Dumped Container!¶

Let's do a little digging where we shouldn't. Go into the `app/cache/dev` directory, where Symfony stores its cache files. In here, there's a file called `appDevDebugProjectContainer.php`. Open it up.

This is *actually* the container class. When you say `$this->container` in your controller, you're getting back an instance of *this* object. Search for the "getEventReportManagerService" function:

```
protected function getEventReportManagerService()
{
    return $this->services['event_report_manager'] =
        new \Yoda\EventBundle\Reporting\EventReportManager(
            $this->get('doctrine.orm.default_entity_manager')
        );
}
```

Internally, when we ask for our service, this is the code that's run. It's not magic, it's just running the exact same PHP code that we had in our controller before registering our class as a service. If we made a change to `services.yml` and refreshed, Symfony would update this file. Pretty amazing.

Chapter 23: Configuration Loading and Type-Hinting

CONFIGURATION LOADING AND TYPE-HINTING¶

So just like with routing files, `services.yml` isn't magically loaded by Symfony: something needs to import it.

When the bundle was generated, an `EventExtension` class was created for you.

This class is mostly useful for third-party bundles, but one thing it does by default is load the `services.yml` file:

```
// src/Yoda/EventBundle/DependencyInjection/EventExtension.php
// ...

public function load(array $configs, ContainerBuilder $container)
{
    // ...
    // this was all generated when we generated the bundle
    $loader->load('services.yml');
}
```

If you don't have this "Extension" class in your bundle, no problem! In fact, delete the entire `DependencyInjection` directory. Now, just import your `services.yml` file from inside `config.yml`:

```
# app/config/config.yml
imports:
    # ...
    - { resource: "@EventBundle/Resources/config/services.yml" }
```

You could also rename `services.yml` to anything else. As you can see, the name isn't important.

Note

The point is that any file that defines a service *must* be imported manually. This can be done via the special "extension" class of a bundle *or* simply by adding it to the `imports` section of `config.yml` or any other configuration file.

Refresh! More CSV Downloading!

Type-Hinting¶

There's one more thing in our service that's bothering me. The first argument to the constructor is the entity manager object, but we're not type-hinting it. Type-hinting is optional, but I like doing it because it gives me better errors and gives me auto-completion in PhpStorm.

So what *is* the class for the entity manager service? One way to find out is to use `container:debug` but pass it the service name:

```
php app/console container:debug doctrine.orm.entity_manager
```

It says that it's just an "alias" for a different service. So let's look up that one:

```
php app/console container:debug doctrine.orm.default_entity_manager
```

Great! Now we can add a type-hint for the argument. And by the way, a lot of times I just guess the class name and let PhpStorm mind trick ... I mean auto-complete the `use` statement for me. It's lazy, but it almost always works:

```php
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

use Doctrine\ORM\EntityManager;

class EventReportManager
{
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }
}
```

If you’re not too comfortable with this, don’t worry. This is optional, but a good practice to get into.

Chapter 24: Dependency Inject All the Things

DEPENDENCY INJECT ALL THE THINGS¶

The CSV returns the `id`, `name` and `time` of each event. Let's pretend that someone is using this to double-check the accuracy of updated events. To make their life easier, I want to also return the URL to each event.

So how do we generate URL's? In `EventController`, we used the `generateUrl` function:

```
$this->generateUrl('event_show', array('slug' => $entity->getSlug()))
```

So let's try putting that into `EventReportManager` and seeing what happens:

```
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

public function getRecentlyUpdatedReport()
{
    // ...

    foreach ($events as $event) {
        $data = array(
            $event->getId(),
            $event->getName(),
            $event->getTime()->format('Y-m-d H:i:s'),
            $this->generateUrl('event_show', array('slug' => $event->getSlug()))
        );

        $rows[] = implode(',', $data);
    }

    return implode("\n", $rows);
}
```

Let's try it. Ah, no download - just an ugly error:

> Call to undefined method YodaEventBundleReportingEventReportManager::generateUrl()

We made this mistake before - `generateUrl` lives in Symfony's `Controller`, and we don't have access to it here. Open up that function to remember what it *actually* does:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php
// ...

public function generateUrl($route, $parameters = array(), $referenceType = UrlGeneratorInterface::ABSOLUTE_PATH)
{
    return $this->container->get('router')
        ->generate($route, $parameters, $referenceType);
}
```

This tells me that if I want to generate a URL, I *actually* need the `router` service. So how can we get the `router` service inside `EventReportManager`? You know the secret: dependency injection.

Add a *second* constructor argument and a second class property:

```php
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

use Doctrine\ORM\EntityManager;
use Symfony\Component\Routing\Router;

class EventReportManager
{
    private $em;

    private $router;

    public function __construct(EntityManager $em, Router $router)
    {
        $this->em = $em;
        $this->router = $router;
    }

    // ...
}
```

This time, I guessed the `router` class name for the type-hint. Now that we have the `router`, just use it in the function:

```php
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

public function getRecentlyUpdatedReport()
{
    // ...

    foreach ($events as $event) {
        $data = array(
            $event->getId(),
            $event->getName(),
            $event->getTime()->format('Y-m-d H:i:s'),
            $this->router->generate('event_show', array('slug' => $event->getSlug()))
        );

        $rows[] = implode(',', $data);
    }

    return implode("\n", $rows);
}
```

Ok, let's test it. Great, now we get a different error:

> Catchable Fatal Error: Argument 2 passed to YodaEventBundleReportingEventReportManager::__construct() must be an instance of SymfonyComponentRoutingRouter, none given

Read it closely. It says that something is calling `__construct` on our class but passing it nothing for the second argument. Of course: we forgot to tell the container about this new argument. Open the `services.yml` file and add a second item to `arguments`:

```yaml
services:
    event_report_manager:
        class: Yoda\EventBundle\Reporting\EventReportManager
        arguments: ["@doctrine.orm.entity_manager", "@router"]
```

Now, we get the download again. Open up the CSV. Hey, we have URL's!

```
5,Darth's Birthday Party!,2014-07-24 12:00:00,/darth-s-birthday-party/show
6,Rebellion Fundraiser Bake Sale!,2014-07-24 12:00:00,/rebellion-fundraiser-bake-sale/show
```

Woops! The URLs aren't helpful unless they're absolute. Pass `true` as the third argument to `generate` to make this happen:

```
// src/Yoda/EventBundle/Reporting/EventReportManager.php
// ...

$data = array(
    $event->getId(),
    $event->getName(),
    $event->getTime()->format('Y-m-d H:i:s'),
    $this->router->generate(
        'event_show',
        array('slug' => $event->getSlug()),
        true
    )
);
```

Download another file and open it up. Perfect!

Here are the *huge* takeaways. When you're in a service and you need to do some work, just find out which service does that work, inject it through the constructor, then use it. You'll use this pattern over and over again. Understand this, and you've mastered the most important concept in Symfony.

Chapter 25: Twig Extensions and Dependency Injection Tags

TWIG EXTENSIONS AND DEPENDENCY INJECTION TAGS¶

We know services. And that makes us really dangerous. Let me show you one of your new tricks.

Twig gives us a ton of built-in functions, filters, tests and other goodies. Everything in Twig - like the `path` function, the `upper` filter and even "tests" like `divisibleby` are loaded into Twig by "extensions", which are basically Twig "plugins".

So can we add our own custom Twig stuff? Of course we can, and it's really fun.

Create a Twig Extension¶

Create a `Twig` directory inside `EventBundle` and a new class called `EventExtension`:

```
// src/Yoda/EventBundle/Twig/EventExtension.php
namespace Yoda\EventBundle\Twig;

class EventExtension
{
}
```

The name and location of this class aren't important and you'll see why. Make the new class extend `Twig_Extension` and add the required `getName` method:

```
// src/Yoda/EventBundle/Twig/EventExtension.php
namespace Yoda\EventBundle\Twig;

class EventExtension extends \Twig_Extension
{
    public function getName()
    {
        return 'event';
    }
}
```

This isn't important - just make sure `getName` returns something unique to your project.

The mission, if you choose to accept it, is to create an `ago` filter: something that'll turn a date into a friendlier phrase like "5 minutes ago".

Use the Non-Existent Filter¶

In `_upcomingEvents.html.twig`, add a new line that takes the `createdAt` time of each event and pushes it through this imaginary `ago` filter:

```
{# src/Yoda/EventBundle/Resources/views/Event/_upcomingEvents.html.twig #}
{# ... #}

<dt>posted:</dt>
<dd>{{ event.createdAt|ago }}</dd>
```

Adding a Custom Filter¶

To add the filter, create a new method called `getFilters` and return an array with a single `ago` entry:

```
// src/Yoda/EventBundle/Twig/EventExtension.php
// ...

public function getFilters()
{
    return array(
        new \Twig_SimpleFilter('ago', array($this, 'calculateAgo')),
    );
}
```

This says: “Hey, whenever someone uses an `ago` filter in Twig, call a `calculateAgo` function”. Create that function and give it a `DateTime` argument:

```
// src/Yoda/EventBundle/Twig/EventExtension.php
// ...

public function calculateAgo(\DateTime $dt)
{
    // todo
}
```

To do the heavy lifting, I’ll use a `DateUtil` class that I have in the code download. Creat a new `Util` directory and paste it there:

```
// src/Yoda/EventBundle/Util/DateUtil.php

namespace Yoda\EventBundle\Util;

use DateTime;

class DateUtil
{
    static public function ago(DateTime $dt)
    {
        // ... check the code download for the source of this class
    }
}
```

Inside `EventExtension`, just call this function statically and return it:

```
// src/Yoda/EventBundle/Twig/EventExtension.php
// ...
use Yoda\EventBundle\Util\DateUtil;
// ...

public function ago(\DateTime $dt)
{
    return DateUtil::ago($dt);
}
```

Tags: Telling Symfony about your Twig Extension¶

Ok, try going to the homepage. It says the filter still doesn’t exist.

We *have* created a valid Twig extension with the filter, but we haven’t actually told Twig about it. Services to the rescue!

First, create a new service for our Twig extension:

```
# src/Yoda/EventBundle/Resources/config/services.yml
services:
    # ...

    twig.event_extension:
        class: Yoda\EventBundle\Twig\EventExtension
        arguments: []
```

Hey, this look familiar! The only difference is that `arguments` is empty, because we don't even have a constructor in this case.

At this point, our Twig extension *is* a service, but Twig still doesn't know about it. Somehow, we need to raise our hand and say "Hey Symfony, this isn't a normal service, it's a Twig Extension!".

Add a `tags` key with a funny-looking `twig.extension` below it:

```
# src/Yoda/EventBundle/Resources/config/services.yml
services:
    # ...

    yoda_event.twig.event_extension:
        class: Yoda\EventBundle\Twig\EventExtension
        arguments: []
        tags:
            - { name: twig.extension }
```

You know how a blog post can have tags? The idea is the same here. When Symfony boots, Twig looks for all services with the `twig.extension` tag and includes those as extensions.

Refresh! The new "posted" text looks fantastic. If you want this functionality in real life, check out the KnpTimeBundle, which is even more powerful.

Note

Want to know more about Twig Extensions? See the official documentation.

More on Tags¶

What other tags are there? Well I'm *so* glad you asked. In the reference section of the docs, we have a fantastic page called The Dependency Injection Tags. If you're doing something really custom, or awesome, in Symfony, you're probably using a dependency injection tag. You won't use them too often, but they're key to unlocking really powerful features.

A very important tag is kernel.event_listener, which allows you to register "hooks" inside Symfony at various stages of the request lifecycle. That topic is for another screencast, but we'll cover a very similar subject next: Doctrine events.

Chapter 26: Doctrine is in your Lifecycle (with Callbacks)

DOCTRINE IS IN YOUR LIFECYCLE (WITH CALLBACKS)¶

Remember when we used StofDoctrineExtensions to set the Event's `slug` for us? That magic works by leveraging one of the most powerful features of Doctrine: events. Doctrine gives us the flexibility to have hooks that are called whenever certain things are done, like when an entity is first persisted, updated, or deleted.

Open up `Event` and remove the `@Gedmo` annotation above `createdAt`. Let's see if we can set this ourselves:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\Column(type="datetime")
 */
private $createdAt;
```

Replace this with a new function called `prePersist` that sets the value:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

public function prePersist()
{
    if (!$this->getCreatedAt()) {
        $this->createdAt = new \DateTime();
    }
}
```

Hey, don't get too excited! This won't work yet, but if we could tell Doctrine to call this before inserting an Event, we'd be golden!

The secret is a called lifecycle callbacks: a fancy word for a function that Doctrine will call when something happens, like when an entity is first inserted.

To enable lifecycle callbacks on an entity, add the `HasLifecycleCallbacks` annotation:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\Table(name="yoda_event")
 * @ORM\Entity(repositoryClass="Yoda\EventBundle\Entity\EventRepository")
 * @ORM\HasLifecycleCallbacks
 */
class Event
{
    // ...
}
```

Now just put a `PrePersist` annotation above our function:

```
// src/Yoda/EventBundle/Entity/Event.php
// ...

/**
 * @ORM\PrePersist
 */
public function prePersist()
{
    if (!$this->getCreated()) {
        $this->createdAt = new \DateTime();
    }
}
```

`PrePersist` is called only when an entity is first inserted, and there are other lifecycle events like `PreUpdate` and `PreRemove`.

Cool, let's give it a test! Reload your fixtures and then query to see the events:

```
php app/console doctrine:fixtures:load
php app/console doctrine:query:sql "SELECT * FROM yoda_event"
```

The `createdAt` column is set so this must be working.

Lifecycle callbacks are brilliant because they're just so easy to setup.

But they have one big limitation. Because the callback is inside an entity, we don't have access to the container or any services. This wasn't a problem here, but what if we needed to access the `router` or the `logger`?

The solution is to use a slight spin on lifecycle callbacks: events.

Chapter 27: Doctrine Event Listeners

DOCTRINE EVENT LISTENERS¶

In episode 2, we created a registration form and manually encoded the user's plain-text password before persisting it. We even duplicated this logic in our fixtures. Shame!

Our goal is to encode the user's password automatically using a Doctrine event listener. These are exactly like a lifecycle callback except that the function that's executed lives *outside* of your entity and inside some other class. Do you think this "other class" will be a service? Of course it will :).

Creating the Event Listener¶

Since I *love* classes so much, create one called `UserListener` in a new `Doctrine` directory of UserBundle:

```
// src/Yoda/UserBundle/Doctrine/UserListener.php
namespace Yoda\UserBundle\Doctrine;

class UserListener
{

}
```

We're going to register this as a service, so the name and location don't matter at all.

Add a `prePersist` method. To prove that this is called, just add a `die` statement:

```
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

class UserListener
{
    public function prePersist()
    {
        die('Something is being inserted!');
    }
}
```

Registering the Listener as a Service¶

Next, let's register this as a service. Hmm, we don't already have a `services.yml` file in UserBundle. Technically, we could just register this in `services.yml` in EventBundle. But to keep things organized, create a new file in UserBundle and configure the service there.

```
# src/Yoda/UserBundle/Resources/config/services.yml
services:
    doctrine.user_listener:
        class: Yoda\UserBundle\Doctrine\UserListener
```

If you think Symfony is going to automatically find this file, you're nuts! Import it manually from your main `config.yml` file:

```
imports:
    # ...
    - { resource: "@UserBundle/Resources/config/services.yml" }
```

Just like with the Twig Extension, our listener *is* a service, but Doctrine doesn't automagically know about it. Let's use another tag, this time called `doctrine.event_listener`:

```
# src/Yoda/UserBundle/Resources/config/services.yml
services:
    doctrine.user_listener:
        class: Yoda\UserBundle\Doctrine\UserListener
        arguments: []
        tags:
            - { name: doctrine.event_listener, event: prePersist }
```

The `name` says we're a listener and `event` tells Doctrine *which* event we're listening to. When Doctrine loads, it looks for all services tagged with `doctrine.event_listener` and makes sure those services are notified on whatever event is specified.

It's the moment of truth! Reload the fixtures:

```
php app/console doctrine:fixtures:load
```

Yes! Our `die` function is hit!

To encode the password, copy in the `encodePassword` from our user fixtures (`LoadUsers.php`) and rename it to `handleEvent`. I'll also make a few other changes, like getting the plain password value off of a `plainPassword` property and setting the encoded password on the user:

```
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...
use Yoda\UserBundle\Entity\User;
// ...

private function handleEvent(User $user)
{
    $plainPassword = $user->getPlainPassword();
    $encoder = $this->container->get('security.encoder_factory')
        ->getEncoder($user);

    $password = $encoder->encodePassword($plainPassword, $user->getSalt());
    $user->setPassword($password);
}
```

This function is *almost* ready.

The Helpful LifecycleEventArgs Callback Argument¶

Whenever Doctrine calls `prePersist`, it passes us a special `LifecycleEventArgs` object. Add an argument for this:

```
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

use Doctrine\ORM\Event\LifecycleEventArgs;

class UserListener
{
    public function prePersist(LifecycleEventArgs $args)
    {
        die('Something is being inserted!');
    }
}
```

We can use this to get the actual object being saved. If that object is an instance of `User`, then we know we want to act on it. If anything else is being saved, we'll just ignore it. This is important because the function is called when *any* entity is saved:

```php
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

public function prePersist(LifecycleEventArgs $args)
{
    $entity = $args->getEntity();
    if ($entity instanceof User) {
        $this->handleEvent($entity);
    }
}
```

Injecting the security.encoder_factory Dependency¶

We're *almost* done. You've probably already noticed that the `$this->container` line won't work here. We don't have a `$container` property - that's something special to controllers and a few other places.

Again *not* a problem! The listener ultimately needs the `security.encoder_factory` service. So let's just inject it. Add a constructor with this as the first argument:

```php
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

use Symfony\Component\Security\Core\Encoder\EncoderFactory;

class UserListener
{
    private $encoderFactory;

    public function __construct(EncoderFactory $encoderFactory)
    {
        $this->encoderFactory = $encoderFactory;
    }
}
```

Use the new property in `handleEvent`:

```php
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

private function handleEvent(User $user)
{
    $plainPassword = $user->getPlainPassword();

    $encoder = $this->encoderFactory
        ->getEncoder($user)
    ;

    $password = $encoder->encodePassword($plainPassword, $user->getSalt());
    $user->setPassword($password);
}
```

The listener is perfect. The last step is to tell the container about the new constructor arugment in `services.yml`:

```yaml
# src/Yoda/UserBundle/Resources/config/services.yml
services:
    doctrine.user_listener:
        class: Yoda\UserBundle\Doctrine\UserListener
        arguments: ["@security.encoder_factory"]
        tags:
            - { name: doctrine.event_listener, event: prePersist }
```

We're ready! Remove all the encoding logic from `LoadUsers` and just set the plain password instead:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

public function load(ObjectManager $manager)
{
    // ...
    // $user->setPassword($this->encodePassword($user, 'darthpass'));
    $user->setPlainPassword('darthpass');

    // ...
    // $admin->setPassword($this->encodePassword($admin, 'waynepass'));
    $admin->setPlainPassword('waynepass');
}
```

Reload the fixtures again!

```
php app/console doctrine:fixtures:load
```

Woh, no errors! Ok, let's login. Hey, that works too! As long as a new `User` has a `plainPassword`, our listener will automatically handle the encoding work for us. With this in place, remove the encoding logic from `RegisterController`.

Chapter 28: Doctrine Listeners on Update

DOCTRINE LISTENERS ON UPDATE¶

But what if a user *updates* their password? Hmm, our listener isn't called on updates, so the encoded password can *never* be updated. Crap!

Add a second tag to `services.yml` to listen on the `preUpdate` event and create the `preUpdate` method by copying from `prePersist`:

```
# src/Yoda/UserBundle/Resources/config/services.yml
services:
    doctrine.user_listener:
        class: Yoda\UserBundle\Doctrine\UserListener
        arguments: ["@security.encoder_factory"]
        tags:
            - { name: doctrine.event_listener, event: prePersist }
            - { name: doctrine.event_listener, event: preUpdate }
```

Add a `die` statement so we can test things:

```
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

public function preUpdate(LifecycleEventArgs $args)
{
    die('UUPPPPPDDAAAAAATING!');

    $entity = $args->getEntity();
    if ($entity instanceof User) {
        $this->handleEvent($entity);
    }
}
```

Also, if the `plainPassword` field isn't set, don't do any work. This will happen if a `User` is being saved, but their password isn't being changed:

```
// src/Yoda/UserBundle/Doctrine/UserListener.php
// ...

private function handleEvent(User $user)
{
    if (!$user->getPlainPasword()) {
        return;
    }

    // ...
}
```

Testing the Update¶

We can't test this easily because we don't have a way to update users yet. No worries. Just open up the play script from episode 1. We already have a user here - just change his plain password and save:

```
// play.php
// ...

use Doctrine\ORM\EntityManager;

$em = $container->get('doctrine')
    ->getEntityManager()
;

$wayne = $em
    ->getRepository('UserBundle:User')
    ->findOneByUsernameOrEmail('wayne');

$wayne->setPlainPassword('new');
$em->persist($user);
$em->flush();
```

Ok, run the play script:

```
php play.php
```

Hmm, it didn't hit our `die` statement. Our listener function wasn't called.

Gotcha 1: Event Listeners don't fire on Unchanged Objects¶

It's a gotcha! The `plainPassword` property isn't saved to Doctrine, but we do *use* it to set the `password` field, which *is* persisted.

The problem is that when we change *only* the `plainPassword` field, the `User` looks "unmodified" to Doctrine. So, instead of calling our listener, it does nothing.

To fix the issue, let's nullify the `password` field whenever `plainPassword` is set:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

public function setPlainPassword($plainPassword)
{
    $this->plainPassword = $plainPassword;

    $this->setPassword(null);

    return $this;
}
```

Since `password` *is* persisted to Doctrine, this is enough to trigger all the normal behavior. Our listener should make sure `password` is set to the encoded value, and not left blank.

Now run the play script again. Great, it hits the `die` statement. Remove that and try it again.

No errors, so let's try to login. Yes!

We just saw prePersist and preUpdate and Doctrine has several other events you can find on their website. Symfony also has events, which are fired at different points during the request-handling process.

Fortunately, Symfony's event system is *very* similar to Doctrine's. Don't you love it when good ideas are shared?

Chapter 29: Keep Going!

KEEP GOING!¶

You're awesome. Seriously. This was probably the *most* important episode yet, and you made it. Congrats!

The first big piece included the two main Doctrine associations: ManyToOne and ManyToMany. We saw how each can have an *optional* inverse side, like the OneToMany side of ManyToOne.

The second *huge* piece was services: how they work and how to create our own. A service is nothing more than a class that does some work. By putting our logic into a service, it makes it reusable, organized, and easier to unit test. When we register a service with Symfony, we teach it how to create a new instance of our object so that we have the convenience of simply getting it out of the container.

We also saw a few "tags", and how you can use them to tell Symfony that your service should be used in some special way.

So what's next? There's always more to learn with Symfony, but we've touched on almost all the most important things by now. In the next, and final screencast in this series, we'll talk about assets, assetic, form customizations and finally deployment and some performance notes.

Seeya next time!