Stripe: Killer Payment Processing + ??? = Profit

With <3 from SymfonyCasts

Chapter 1: Your Stripe Dashboard

So you want to handle payment on the web? You *brave*, foolish soul :).

Nah, it's fine - these days, handling credit card payments is a *blast*. Especially with Stripe - an awesome service we've used on KnpU for years.

But let's be real: you're dealing with people's money, so don't muck it up! If you screw up or do something insecure, there could be real consequences. You will *at least* have an angry customer. So I guess this tutorial is all about accepting money and having *happy* customers.

So let's build a real-life, robust payment system so that when things go wrong - because they will - we fail gracefully, avoid surprises and make happy customers.

Code Along with Me. Do it!

As always, I beg you, I *implore* you, to code along with me! To do that, download the course code on this page, unzip it, and move into the start/ directory. That will give you the same code I have here.

This is a Symfony project, but we'll avoid going too deep into that stuff because I want to focus on Stripe. Inside, open the README.md file and follow the setup details to get the project running. The last step is to open your favorite console app, move into the directory, and run:

```
$ php bin/console server:run
```

to start up the built-in web server.

Our Great Idea: Sheep Shear Club

But before you start collecting any money, you need to come up with that next, *huge* idea. And here at KnpUniversity, we're convinced we've uncovered the next tech unicorn.

Ready to find out what it is? Open your browser, and go to:

```
http://localhost:8000
```

That's right: welcome to The Sheep Shear Club, your one-stop shop for artisanal shearing accessories for the most dapper sheep. Purchase cutting-edge individual products - like one of our After-Shear scents - or have products delivered directly to your farm with a monthly subscription.

Gosh, it's *shear* luck that we got to this idea first. Once we finish coding the checkout, our competition will be feeling *sheepish*.

But the site is pretty simple: we have a login page - the password is breakingbaad. After you login, you can add items to your cart and they'll show up on the checkout page. But notice, there is *no* checkout form yet. That's our job.

Getting to Know your Stripe Dashboard

The first step towards that is to sign up with a fancy new account on Stripe. Once you're in, you'll see this: your new e-commerce best friend: the Stripe dashboard.

There is *a lot* here, but right now I want you to notice that there are two environments: "test" and "live". These are like two totally separate databases full of orders, customers and more, and you can just switch between them to see your data.

Also, once you login, when you read the Stripe documentation, it will actually pre-fill *your* account's API keys into code examples.

Let's use those docs to put in our checkout form!

Chapter 2: Embedded Checkout Form

Click on the documentation link at the top, and then click Embedded Form. There are two ways to build a checkout-form: the easy & lazy way - via an *embedded form* that Stripe builds for you - or the harder way - with an HTML form that you build yourself. Our sheep investors want us to hoof-it and get this live, so let's do the easy way first - and switch to a custom form later.

Getting the Form Script Code

To get the embedded form, copy the form code. Then, head to the app and open the app/Resources/views/order/checkout.html.twig file. This is the template for the checkout page.

At the bottom, I already have a spot waiting for the checkout form. Paste the code there:

52 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
                    <script
                        src="https://checkout.stripe.com/checkout.js" class="stripe-button"
                        data-key="pk_test_HxZzNHy8LImKK9LDtgMDRBwd"
                        data-amount="999"
                        data-name="Dollar Shear Club"
                        data-description="Widget"
                        data-image="/img/documentation/checkout/marketplace.png"
                        data-locale="auto">
                    </script>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Oh! And as promised: this pk_test value is the *public* key from *our* test environment.

Your Stripe Public and Private Keys

Let me show you what I mean: in Stripe, open your "Account Settings" on the top and then select "API Keys". Each environment - Test and Live - have their own two keys: the secret key and the publishable or public key. Right now, we're using the public key for the test environment - so once we get this going, orders will show up there. After we deploy, we'll need to switch to the Live environment keys.

Oh, and, I think it's obvious - but these secret keys need to be kept secret. The *last* thing you should do is create a screencast and publish them to the web. Oh crap.

Hey, A Checkout Form

But anyways, without doing any more work, go back to the browser and refresh the page. Hello checkout button! And hello checkout form! Obviously, $9.99 isn't the right price, for these amazing sheep accessories.

To fix that, head back to the template. Everything about the form is controlled with these HTML attributes. Obviously, the most important one is amount. Set it to {{ cart.total }} - cart is a variable I've passed into the template - then the important part: * 100:

```
53 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
                    <script
... lines 38 - 39
                        data-amount="{{ cart.total * 100 }}"
... lines 41 - 45
                    </script>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Whenever you talk about *amounts* in Stripe, you use the *smallest* denomination of the currency, so cents in USD. If you need to charge the user $5, then tell Stripe to charge them an amount of 500 cents.

Then, fill in anything else that's important to you, for example, data-image. I'll set this to our logo:

```
53 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
                    <script
... lines 38 - 42
                        data-image="{{ asset('images/logo.png') }}"
... lines 44 - 45
                    </script>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Checking out with a Test Card

Refresh to reflect the new settings. The total should be $62, and it is! Because we're using the test environment, Stripe gives us fake, test cards we can use to checkout. I'll show you others later - but to checkout successfully, use 4242 4242 4242 4242. You can use any valid future expiration and any CVC.

Ok, moment of truth: hit pay!

It worked! I think... Wait, what just happened? Well, a *really* important step just happened - a step that's *core* to how Stripe checkout works.

The Stripe Checkout Token

First, credit card information is *never* sent to our servers... which is the *greatest* news ever from a security standpoint. I do *not* want to handle your CC number: this would *greatly* increase the security requirements on my server.

Instead, when you hit "Pay", this sends the credit card information to *Stripe* directly, via AJAX. If the card is valid, it sends back a token string, which *represents* that card. The Stripe JS puts that token into the form as an hidden input field and then submits the form like normal to our server. So the *only* thing that's sent to our server is this token. The customer has *not* been charged yet, but with a little more work - we can fetch that token in our code and ask Stripe to charge that credit card.

Fetching the Stripe Token

Let's go get that token on the server. Open up src/AppBundle/Controller/OrderController.php and find checkoutAction():

40 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 9
class OrderController extends BaseController
{
... lines 12 - 25
    /**
     * @Route("/checkout", name="order_checkout")
     * @Security("is_granted('ROLE_USER')")
     */
    public function checkoutAction()
    {
        $products = $this->get('shopping_cart')->getProducts();

        return $this->render('order/checkout.html.twig', array(
            'products' => $products,
            'cart' => $this->get('shopping_cart')
        ));

    }
}
```

This controller renders the checkout page. And because the HTML form has action="":

53 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
... lines 37 - 46
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

When Stripe submits the form, it submits *right* back to this same URL and controller.

To fetch the token, add a Request argument, and make sure you have the use statement on top:

```
47 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 8
use Symfony\Component\HttpFoundation\Request;

class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 45
    }
}
```

Then, inside the method, say if ($request->isMethod('POST'), then we know the form was just submitted. If so, dump($request->get('stripeToken')):

```
47 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 10
class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
        $products = $this->get('shopping_cart')->getProducts();

        if ($request->isMethod('POST')) {
            $token = $request->request->get('stripeToken');

            dump($token);
        }
... lines 40 - 45
    }
}
```

If you read Stripe's documentation, that's the name of the hidden input field.

Try it out! Refresh and fill in the info again: use your trusty fake credit card number, some fake data and Pay. The form submits and the page refreshes. But thanks to the dump() function, hover over the target icon in the web debug toolbar. Perfect! We were able to fetch the token.

In a second, we're going to send this back to Stripe and ask them to actually charge the credit card it represents. But before we do that, head back to the Stripe dashboard.

Stripe shows you a log of pretty much *everything* that happens. Click the Logs link: these are *all* the interactions we've had with Stripe's API, including a few from before I hit record on the screencast. Click the first one: this is the AJAX request that the JavaScript made to Stripe: it sent over the credit card information, and Stripe sent back the token. If I search for the token that was just dumped, it matches.

Ok, let's use that token to charge our customer.

Chapter 3: Charge It (The Stripe PHP SDK)

Stripe has a nice, RESTful API, and you're going to spend a lot of time talking with it. Google for "Stripe API docs" to find this amazing page. You can set this as your new homepage: it describes *every* endpoint: how to create charges, customers and a lot of other things we're going to talk about.

But first, make sure that you select PHP on the top right. Thanks to this, the docs will show you code snippets in PHP. And those code snippets will use Stripe's PHP SDK library. Google for that and open its Github Page.

First, let's get this guy installed. Copy the composer require line, move over to your terminal, open a new tab and paste that:

```
$ composer require stripe/stripe-php
```

While we're waiting for Jordi to finish, let's keep going.

Using the Token to Create a Charge

To actually charge a user, we need to... well, create a Stripe *charge*. In the Stripe API, click "Charges" on the left and find Create a Charge.

Hey! It wrote the code *for* us. Copy the code block on the right. Now, go back to OrderController and first, create a new $token variable and set it to the stripeToken POST parameter. Now, paste that code:

58 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 34
        if ($request->isMethod('POST')) {
            $token = $request->request->get('stripeToken');

            \Stripe\Stripe::setApiKey("XXX_PRIVATEKEY_XXX");
            \Stripe\Charge::create(array(
              "amount" => $this->get('shopping_cart')->getTotal() * 100,
              "currency" => "usd",
              "source" => $token,
              "description" => "First test charge!"
            ));
... lines 45 - 49
        }
... lines 51 - 56
    }
}
```

Let's go check on Composer. It's *just* finishing - perfect! My editor now sees all these new Stripe classes.

See that API key?

```
58 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 10
class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 34
        if ($request->isMethod('POST')) {
... lines 36 - 37
            \Stripe\Stripe::setApiKey("XXX_PRIVATEKEY_XXX");
... lines 39 - 49
        }
... lines 51 - 56
    }
}
```

Once again, this is a real key from *our* account in the test environment. This time, it's the *secret* key. The public key is the one in our template:

```
53 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
                    <script
... line 38
                        data-key="pk_test_HxZzNHy8LImKK9LDtgMDRBwd"
... lines 40 - 45
                    </script>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Update charge details with our *real* information. To get the total, I'll fetch a service I created for the project called shopping_cart, call getTotal() on this, and then multiply it by 100:

58 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 10
class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 34
        if ($request->isMethod('POST')) {
... lines 36 - 38
            \Stripe\Charge::create(array(
              "amount" => $this->get('shopping_cart')->getTotal() * 100,
... lines 41 - 43
            ));
... lines 45 - 49
        }
... lines 51 - 56
    }
}
```

For source, replace this fake token with the *submitted* $token variable:

58 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 10
class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 34
        if ($request->isMethod('POST')) {
            $token = $request->request->get('stripeToken');
... lines 37 - 38
            \Stripe\Charge::create(array(
... lines 40 - 41
              "source" => $token,
... line 43
            ));
... lines 45 - 49
        }
... lines 51 - 56
    }
}
```

The token basically represents the credit card that was just sent. We're saying: Use *this* card as the source for this charge. And then, put whatever you want for description, like "First test charge":

58 lines | src/AppBundle/Controller/OrderController.php

```
// ... lines 1 - 10
class OrderController extends BaseController
{
// ... lines 13 - 30
    public function checkoutAction(Request $request)
    {
// ... lines 33 - 34
        if ($request->isMethod('POST')) {
// ... lines 36 - 38
            \Stripe\Charge::create(array(
// ... lines 40 - 42
              "description" => "First test charge!"
            ));
// ... lines 45 - 49
        }
// ... lines 51 - 56
    }
}
```

When this code runs, it will make an API request to Stripe. If that's successful, the user will be charged. If something goes wrong, Stripe will throw an Exception. More on that later.

Cleaning up after Checkout

But before we try it, we need to finish up a few application-specific things. For example, after check out, we need to empty the shopping cart. The products are great, but the customer probably doesn't want to buy them twice in a row:

58 lines | src/AppBundle/Controller/OrderController.php

```
// ... lines 1 - 10
class OrderController extends BaseController
{
// ... lines 13 - 30
    public function checkoutAction(Request $request)
    {
// ... lines 33 - 34
        if ($request->isMethod('POST')) {
// ... lines 36 - 38
            \Stripe\Charge::create(array(
// ... lines 40 - 43
            ));

            $this->get('shopping_cart')->emptyCart();
// ... lines 47 - 49
        }
// ... lines 51 - 56
    }
}
```

Next, I want to show a success message to the user. To do that in Symfony, call $this->addFlash('success', 'Order Complete! Yay!'):

```
58 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 10
class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 34
        if ($request->isMethod('POST')) {
... lines 36 - 45
            $this->get('shopping_cart')->emptyCart();
            $this->addFlash('success', 'Order Complete! Yay!');

            return $this->redirectToRoute('homepage');
        }
... lines 51 - 56
    }
}
```

And finally, you should *definitely* redirect the page somewhere. I'll use redirectToRoute() to send the user to the homepage.

That is it. Now for the *real* moment of truth. Hit enter to reload our page without submitting, put in the fake credit card, any date, any CVC, and...

Tip

If you get some sort of API or connection, you may need to upgrade some TLS security settings.

Hey! Okay. No errors. That *should* mean it worked. How can we know? Go check out the Stripe Dashboard. This time, click "Payments". And there's our payment for $62. You can even see all the information that was used.

Congratulations guys! You just added a checkout to your site in 15 minutes. Now let's make this thing rock-solid.

Chapter 4: Hide Those Private Keys

We already know that our Stripe account has *two* environments, and each has its *own* two keys. This means that when we deploy, we'll need to update our code to use these *Live* keys, instead of the ones from the Test environment.

Well... that's going to be a bummer: the public key is hard-coded right in the middle of my template:

53 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
                    <script
... line 38
                        data-key="pk_test_HxZzNHy8LImKK9LDtgMDRBwd"
... lines 40 - 45
                    </script>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

And the private one is stuck in the center of a controller:

58 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 10
class OrderController extends BaseController
{
... lines 13 - 30
    public function checkoutAction(Request $request)
    {
... lines 33 - 34
        if ($request->isMethod('POST')) {
... lines 36 - 37
            \Stripe\Stripe::setApiKey("XXX_PRIVATEKEY_XXX");
... lines 39 - 49
        }
... lines 51 - 56
    }
}
```

If you *love* editing random files whenever you deploy, then this is perfect! Have fun!

But for the rest of us, we need to move these keys to a central configuration file so they're easy to update on deploy. We also need to make sure that we don't commit this private key to our Git repository... ya know... because it's private - even though I keep showing you mine.

Quick! To a Configuration File!

How you do this next step will vary for different frameworks, but is philosophically always the same. In Symfony, we're going to move our keys to a special parameters.yml file, because our project is setup to *not* commit this to Git.

Add a stripe_secret_key config and set its value to the key from the controller:

22 lines | app/config/parameters.yml.dist

```
... lines 1 - 3
parameters:
... lines 5 - 19
    stripe_secret_key: XXX_PRIVATEKEY_XXX
... lines 21 - 22
```

Then add stripe_public_key and set that to the one from the template:

22 lines | app/config/parameters.yml.dist

```
... lines 1 - 3
parameters:
... lines 5 - 20
    stripe_public_key: YYY_PUBLISHABLE_KEY_YYY
```

In Symfony, we also maintain this other file - parameters.yml.dist - as a *template* for the original, uncommitted file. This one *is* committed to the repository. Add the keys here too, but give them fake values.

Using the Parameters

Now that these are isolated in parameters.yml, we can take them out of our code. In the controller, add $this->getParameter('stripe_secret_key'):

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 38
            \Stripe\Stripe::setApiKey($this->getParameter('stripe_secret_key'));
... lines 40 - 70
        }
... lines 72 - 78
    }
}
```

Next, pass a new stripe_public_key variable to the template set to $this->getParameter('stripe_public_key'):

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 72
        return $this->render('order/checkout.html.twig', array(
... lines 74 - 75
            'stripe_public_key' => $this->getParameter('stripe_public_key')
        ));

    }
}
```

Finally, in the template - render that new variable:

53 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 8 - 34
            <div class="col-xs-12 col-sm-6">
                <form action="" method="POST">
                    <script
... line 38
                        data-key="{{ stripe_public_key }}"
... lines 40 - 45
                    </script>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Make sure we didn't break anything by finding a product and adding it to the cart. The fact that this "Pay with Card" shows up means things are probably OK.

This was a small step, but don't mess it up! If that secret key becomes *not* so secret, sheep-zombies will attack.

Chapter 5: Stripe Customers + Our Users

As you can see, there are a *lot* of *objects* that you can interact with in Stripe's API. We've only talked about one: Charge. And it's enough to collect money.

But to create a *really* nice system, you need to talk about the Customer object. Customers give you *three* big super-powers.

First, if the same customer comes back over-and-over again, right now, they need to enter their credit card information *every* time. But with a Customer, you can *store* cards in Stripe and charge them using that. Second, all the charges for a customer will show up in one spot in Stripe's dashboard, instead of all over the place. And third, customers are needed to process *subscription* payments - something we'll talk about in the next Stripe course.

Storing stripeUserId on the User Table

So here's the goal: instead of creating random Charge objects, let's create a Customer object in Stripe and *then* charge that Customer. We also need to save the customer id to our user table so that when that user comes back in the future, we'll know that they are already a customer in Stripe.

In this project, the name of the user table is fos_user, and it contains *just* some basic fields, like email, username and few others related to things like resetting your password.

Let's add a new column called stripeUserId. To do that, open a class in AppBundle called User. Create a new private property called stripeCustomerId:

```
41 lines | src/AppBundle/Entity/User.php
 ... lines 1 - 11
class User extends BaseUser
{
 ... lines 14 - 23
    private $stripeCustomerId;
 ... lines 25 - 39
}
```

Above that, we're going to use annotations to say @ORM\Column(type="string") to create a varchar column. Let's even add a unique index on this field and add nullable=true to allow the field to be empty in the database:

```
41 lines | src/AppBundle/Entity/User.php
 ... lines 1 - 11
class User extends BaseUser
{
 ... lines 14 - 20
    /**
     * @ORM\Column(type="string", unique=true, nullable=true)
     */
    private $stripeCustomerId;
 ... lines 25 - 39
}
```

At the bottom of the class, I'll use the "Code"->"Generate" menu - or Command+N on a Mac - to generate the getters and setters for the new property:

41 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 11
class User extends BaseUser
{
... lines 14 - 30
    public function getStripeCustomerId()
    {
        return $this->stripeCustomerId;
    }

    public function setStripeCustomerId($stripeCustomerId)
    {
        $this->stripeCustomerId = $stripeCustomerId;
    }
}
```

Now that our PHP code is updated, we need to actually add the new column to our table. Since this project uses Doctrine migrations, open a new tab and run:

```
$ php bin/console doctrine:migrations:diff
```

All that did was create a new file that contains the raw SQL needed to add this new stripe_customer_id column:

37 lines | app/DoctrineMigrations/Version20160709170231.php

```
... lines 1 - 10
class Version20160709170231 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('ALTER TABLE fos_user ADD stripe_customer_id VARCHAR(255) DEFAULT NULL');
        $this->addSql('CREATE UNIQUE INDEX UNIQ_957A6479708DC647 ON fos_user (stripe_customer_id)');
    }
... lines 24 - 27
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('DROP INDEX UNIQ_957A6479708DC647 ON fos_user');
        $this->addSql('ALTER TABLE fos_user DROP stripe_customer_id');
    }
}
```

To execute that, run another command:

```
$ php bin/console doctrine:migrations:migrate
```

Perfect! Back in SQL, you can see the fancy new column on the table.

We are ready to create Stripe customers.

Chapter 6: We <3 Creating Stripe Customers

Head back to OrderController. Create a $user variable set to $this->getUser():

```
80 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 38
            \Stripe\Stripe::setApiKey($this->getParameter('stripe_secret_key'));

            /** @var User $user */
            $user = $this->getUser();
... lines 43 - 70
        }
... lines 72 - 78
    }
}
```

This is the User object for *who* is currently logged in. I'll add some inline documentation to show that.

When the user submits the payment form, there are *two* different scenarios. First, if (!$user->getStripeCustomerId()), then this is a first-time buyer, and we need to create a new Stripe Customer for them:

```
80 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 40
            /** @var User $user */
            $user = $this->getUser();
            if (!$user->getStripeCustomerId()) {
... lines 44 - 57
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

To do that, go back to their API documentation and find Create A Customer. Oh hey, it wrote the code for us again! Steal it! And paste it right inside the if statement:

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
                $customer = \Stripe\Customer::create([
                    'email' => $user->getEmail(),
                    'source' => $token
                ]);
... lines 48 - 57
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

Customer has a lot of fields, but most are optional. Let's set email to $user->getEmail():

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
                $customer = \Stripe\Customer::create([
                    'email' => $user->getEmail(),
... line 46
                ]);
... lines 48 - 57
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

So we can easily look up a user in Stripe's dashboard later.

The *really* important field is source. This refers to the *payment source* - so credit or debit card in our case - that you want to attach to the customer. Set this to the $token variable:

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
                $customer = \Stripe\Customer::create([
... line 45
                    'source' => $token
                ]);
... lines 48 - 57
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

This is huge: it will attach that card to their account, and allow us - if we want to - to charge them using that same card in the future.

Set this call to a new $customer variable: the create() method returns a Stripe\Customer object:

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
                $customer = \Stripe\Customer::create([
                    'email' => $user->getEmail(),
                    'source' => $token
                ]);
... lines 48 - 57
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

And we like that because *this* object has an id property.

To save that on our user record, say $user->setStripeCustomerId($customer->id):

80 lines | src/AppBundle/Controller/OrderController.php

```php
// ... lines 1 - 11
class OrderController extends BaseController
{
// ... lines 14 - 31
    public function checkoutAction(Request $request)
    {
// ... lines 34 - 35
        if ($request->isMethod('POST')) {
// ... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
                $customer = \Stripe\Customer::create([
                    'email' => $user->getEmail(),
                    'source' => $token
                ]);
                $user->setStripeCustomerId($customer->id);
// ... lines 49 - 57
            }
// ... lines 59 - 70
        }
// ... lines 72 - 78
    }
}
```

Then, I'll use Doctrine to run the UPDATE query to the database:

80 lines | src/AppBundle/Controller/OrderController.php

```php
// ... lines 1 - 11
class OrderController extends BaseController
{
// ... lines 14 - 31
    public function checkoutAction(Request $request)
    {
// ... lines 34 - 35
        if ($request->isMethod('POST')) {
// ... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
// ... lines 44 - 47
                $user->setStripeCustomerId($customer->id);

                $em = $this->getDoctrine()->getManager();
                $em->persist($user);
                $em->flush();
// ... lines 53 - 57
            }
// ... lines 59 - 70
        }
// ... lines 72 - 78
    }
}
```

If you're not using Doctrine, just make sure to update the user record in the database however you want.

Fetching the Existing Customer Object

Now, add the else: this means the user *already* has a Stripe customer object. Repeat customer! Instead of creating a new one, just fetch the customer with \Stripe\Customer::retrieve() and pass it $user->getStripeCustomerId():

```
80 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
... lines 44 - 52
            } else {
                $customer = \Stripe\Customer::retrieve($user->getStripeCustomerId());
... lines 55 - 57
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

Since this user is already in Stripe, we *might* eventually re-work our checkout page so that they *don't* need to re-enter their credit card. But, we haven't done that yet. And since they just submitted *fresh* card information, we should *update* their account with that. After all, this might be a different card than what they used the first time they ordered.

To do that, update the source field: set it to $token. To send that update to Stripe, call $customer->save():

```
80 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
... lines 44 - 52
            } else {
                $customer = \Stripe\Customer::retrieve($user->getStripeCustomerId());

                $customer->source = $token;
                $customer->save();
            }
... lines 59 - 70
        }
... lines 72 - 78
    }
}
```

So in *both* situations, the token will now be attached to the customer that's associated with our user. Phew!

Charging the User

The last thing we need to update is the Charge: instead of passing source, charge the *customer* instead. Set 'customer' => $user->getStripeCustomerId():

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 59
            \Stripe\Charge::create(array(
... lines 61 - 62
                "customer" => $user->getStripeCustomerId(),
... line 64
            ));
... lines 66 - 70
        }
... lines 72 - 78
    }
}
```

We're no long saying "Charge this credit card", we're saying "Charge this customer, using whatever credit card they have on file".

Ok, time to try it out! Go back and reload this page. Run through the checkout with our fake data and hit Pay. Hey, hey - no errors!

So go check your Stripe dashboard. Under Payments, you should see this new charge. And if you click into it, it is now associated with a *customer*. Success! The customer page shows even more information: the attached card, any past payments and eventually subscriptions. This is one big step forward.

Copy the customer's id and query for that on our fos_user table. Yes, it *did* update!

Since adding a customer went so well, let's talk about *invoices*.

Chapter 7: Stripe Invoices

The first two important objects in Stripe are Charge and Customer.

Let's talk about the *third* important object: Invoice. Here's the idea: right now, we simply charge the Customer. But instead of doing that, we *could* add invoice items to the Customer, create an Invoice for those items, and then pay that Invoice.

To the user, this feels the same. But in Stripe, instead of having a charge, you will have an Invoice full of invoice items, and a charge to pay that invoice. Why do we care? Well first, it let's you have detailed line-items - like two separate items if our customer orders 2 products.

And second, invoices are *central* to handling *subscriptions*. In fact, you'll find the Invoice API documentation under the subscription area. But, it can be used for any charges.

Creating & Paying the Invoice

Let's hook this up. First, instead of creating a Stripe Charge, create a Stripe InvoiceItem:

85 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 59
            \Stripe\InvoiceItem::create(array(
              "amount" => $this->get('shopping_cart')->getTotal() * 100,
              "currency" => "usd",
              "customer" => $user->getStripeCustomerId(),
              "description" => "First test charge!"
            ));
... lines 66 - 75
        }
... lines 77 - 83
    }
}
```

But all the data is the same.

Below that, add $invoice = \Stripe\Invoice::create() and pass that an array with customer set to $user->getStripeCustomerId():

85 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 59
            \Stripe\InvoiceItem::create(array(
              "amount" => $this->get('shopping_cart')->getTotal() * 100,
              "currency" => "usd",
              "customer" => $user->getStripeCustomerId(),
              "description" => "First test charge!"
            ));
            $invoice = \Stripe\Invoice::create(array(
                "customer" => $user->getStripeCustomerId()
            ));
... lines 69 - 75
        }
... lines 77 - 83
    }
}
```

Finally, add $invoice->pay():

85 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 65
            $invoice = \Stripe\Invoice::create(array(
                "customer" => $user->getStripeCustomerId()
            ));
            // guarantee it charges *right* now
            $invoice->pay();
... lines 71 - 75
        }
... lines 77 - 83
    }
}
```

Let's break this down. The first part creates an InvoiceItem in Stripe, but nothing is charged yet. Then, when you create an Invoice, Stripe looks for all unpaid invoice items and attaches them to that Invoice. The last line charges the customer to pay that invoice's balance.

Usually, when you create an Invoice, Stripe will charge the customer immediately. But, if you have web hooks setup - something we'll talk about in the second course - that will delay charging the user by 1 hour. Calling ->pay() guarantees that this happens *right* now.

Ok, go back and try this out. Find a great and high-quality product, and add it to the cart. Checkout using your favorite fake credit card and fake information.

Looks like it worked! And since this user *already* is a Stripe customer, refresh that customer's page in Stripe. Check this out! We have *two* payments and we can see the Invoice. If you click that, the Invoice has 1 line item *and* a related Charge object.

Tip

If all charges belong to an invoice, you can use Stripe's API to retrieve your customer's past invoices and render them as a receipt.

One InvoiceItem per Product

This now gives us more flexibility. Since sheep love to shop, they'll often buy *multiple* products. In fact, let's go buy some shears, and some Sheerly Conditioned. If we checked out right now, this would show up as one giant line item for $106.00 on the Invoice. We can do better than that.

In OrderController, around the InvoiceItem part, add a foreach, over $this->get('shopping_cart')->getProducts() as $product. In other words, let's loop over all the actual products in our cart and create a separate InvoiceItem for each. All we need to do is change the amount to be: $product->getPrice() * 100. We can even improve the description: set it to $product->getName():

87 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 59
            foreach ($this->get('shopping_cart')->getProducts() as $product) {
                \Stripe\InvoiceItem::create(array(
                    "amount" => $product->getPrice() * 100,
                    "currency" => "usd",
                    "customer" => $user->getStripeCustomerId(),
                    "description" => $product->getName()
                ));
            }
... lines 68 - 77
        }
... lines 79 - 85
    }
}
```

Now, if we eventually send the user a receipt, it's going to be very easy to look on this Stripe invoice and see exactly what we charged them for.

Test time! Put in our awesome fake information, hit ENTER... and no errors.

In Stripe, click back to the customer page and find the new invoice - for $106. Click on that. Yes! 2 crystal clear invoice line items.

So yes, you can just charge customers. But if you create an Invoice with detailed line items, you're going to have a much better accounting system in the long run.

Chapter 8: Centralize your Stripe Code

Stripe's API is really organized. Our code that *talks* to it is getting a little crazy, unless you like long, procedural code that you can't re-use. Please tell me that's not the case.

Let's get this organized! At the very least, we should do this because eventually we're going to need to re-use some of this logic - particularly with subscriptions.

Here's the goal of the next few minutes: move each *thing* we're doing in the controller into a set of nice, re-usable functions. To do that, inside AppBundle, create a new class called StripeClient:

```
9 lines | src/AppBundle/StripeClient.php
... lines 1 - 2
namespace AppBundle;

class StripeClient
{

}
```

Make sure this has the AppBundle namespace. We're going to fill this with functions that work with Stripe, like createCustomer() or updateCustomerCard().

Moving createCustomer()

In the controller, the first thing we do is create a Customer:

```
87 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 42
            if (!$user->getStripeCustomerId()) {
                $customer = \Stripe\Customer::create([
                    'email' => $user->getEmail(),
                    'source' => $token
                ]);
                $user->setStripeCustomerId($customer->id);
... lines 49 - 57
            }
... lines 59 - 77
        }
... lines 79 - 85
    }
}
```

In StripeClient, add a new createCustomer() method that will accept the User object which should be associated with the customer, and the $paymentToken that was just submitted:

31 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 7
class StripeClient
{
... lines 10 - 16
    public function createCustomer(User $user, $paymentToken)
    {
... lines 19 - 28
    }
}
```

Copy the logic from the controller and paste it here. Update $token to $paymentToken. Then, return the $customer at the bottom, just in case we need it:

31 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 7
class StripeClient
{
... lines 10 - 16
    public function createCustomer(User $user, $paymentToken)
    {
        $customer = \Stripe\Customer::create([
            'email' => $user->getEmail(),
            'source' => $paymentToken,
        ]);
        $user->setStripeCustomerId($customer->id);

        $this->em->persist($user);
        $this->em->flush($user);

        return $customer;
    }
}
```

You'll see me do with this most functions in this class.

The only problem is with the entity manager - the code used to update the user record in the database. The way we fix this is a bit specific to Symfony. First, add a public function __construct() with an EntityManager $em argument. Set this on a new $em property:

31 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 5
use Doctrine\ORM\EntityManager;

class StripeClient
{
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }
... lines 16 - 29
}
```

Down below, just say $em = $this->em:

31 lines | src/AppBundle/StripeClient.php

```php
... lines 1 - 7
class StripeClient
{
... lines 10 - 16
    public function createCustomer(User $user, $paymentToken)
    {
... lines 19 - 24
        $this->em->persist($user);
        $this->em->flush($user);
... lines 27 - 28
    }
}
```

Registering the Service

To use the new function in our controller, we need to register it as a service. Open up app/config/services.yml. Add a service called stripe_client, set its class key to AppBundle\StripeClient and set autowire to true:

14 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
... lines 7 - 10
    stripe_client:
        class: AppBundle\StripeClient
        autowire: true
```

With that, Symfony will guess the constructor arguments to the object.

If you're not coding in Symfony, that's OK! Do whatever you need to in order to have a set of re-usable functions for interacting with Stripe.

In the controller, clear out all the code in the if statement, and before it, add a new variable called $stripeClient set to $this->get('stripe_client'):

80 lines | src/AppBundle/Controller/OrderController.php

```php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
            $token = $request->request->get('stripeToken');

            \Stripe\Stripe::setApiKey($this->getParameter('stripe_secret_key'));

            $stripeClient = $this->get('stripe_client');
... lines 42 - 70
        }
... lines 72 - 78
    }
}
```

This will be an instance of that StripeClient class.

In this if, call $stripeClient->createCustomer() and pass it the $user object and the $token:

```
80 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 40
            $stripeClient = $this->get('stripe_client');
... lines 42 - 43
            if (!$user->getStripeCustomerId()) {
                $stripeClient->createCustomer($user, $token);
            } else {
... lines 47 - 50
            }
... lines 52 - 70
        }
... lines 72 - 78
    }
}
```

Done.

Moving updateCustomerCard()

Let's keep going!

The *second* piece of logic is responsible for updating the card on an existing customer. In StripeClient, add a public function updateCustomerCard() with a User $user whose related Customer should be updated, and the new $paymentToken:

```
39 lines | src/AppBundle/StripeClient.php
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 7
class StripeClient
{
... lines 10 - 30
    public function updateCustomerCard(User $user, $paymentToken)
    {
... lines 33 - 36
    }
}
```

Copy logic from the controller and past it here. Update $token to $paymentToken:

```
39 lines | src/AppBundle/StripeClient.php
... lines 1 - 7
class StripeClient
{
... lines 10 - 30
    public function updateCustomerCard(User $user, $paymentToken)
    {
        $customer = \Stripe\Customer::retrieve($user->getStripeCustomerId());

        $customer->source = $paymentToken;
        $customer->save();
    }
}
```

Go copy the logic from the controller, and paste it here. Update $token to $paymentToken.

In OrderController, call this with $stripeClient->updateCustomerCard() passing it $user and $token:

```
77 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 40
            $stripeClient = $this->get('stripe_client');
... lines 42 - 43
            if (!$user->getStripeCustomerId()) {
                $stripeClient->createCustomer($user, $token);
            } else {
                $stripeClient->updateCustomerCard($user, $token);
            }
... lines 49 - 67
        }
... lines 69 - 75
    }
}
```

Now the StripeClient class is getting dangerous!

Always setting the API Key

But, there's one small problem. This *will* work now, but look at the setApiKey() method call that's above everything:

77 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 38
            \Stripe\Stripe::setApiKey($this->getParameter('stripe_secret_key'));
... lines 40 - 67
        }
... lines 69 - 75
    }
}
```

We *must* call this before we make any API calls to Stripe. So, if we tried to use the StripeClient somewhere *else* in our code, but we forgot to call this line, we would have *big* problems.

Instead, I want to *guarantee* that if somebody calls a method on StripeClient, setApiKey() will always be called first. To do that, copy that line, delete it and move it into StripeClient's __construct() method.

Symfony user's will know that the getParameter() method won't work here. To fix that, add a new *first* constructor argument called $secretKey. Then, use that:

41 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 7
class StripeClient
{
... lines 10 - 11
    public function __construct($secretKey, EntityManager $em)
    {
... lines 14 - 15
        \Stripe\Stripe::setApiKey($secretKey);
    }
... lines 18 - 39
}
```

To tell Symfony to pass this, go back to services.yml and add an arguments key with one entry: %stripe_secret_key%:

15 lines | app/config/services.yml

```
... lines 1 - 10
    stripe_client:
... line 12
        arguments: ['%stripe_secret_key%']
... line 14
```

Thanks to auto-wiring, Symfony will pass the stripe_secret_key parameter as the first argument, but then autowire the second, EntityManager argument.

The end-result is this: when our StripeClient object is created, the API key is set immediately.

Moving Invoice Logic

Ok, the hard stuff is behind us: let's move the last two pieces of logic: creating an InvoiceItem and creating an Invoice. In StripeClient, add public function createInvoiceItem() with an $amount argument, the $user to attach it to and a $description:

```
65 lines | src/AppBundle/StripeClient.php
... lines 1 - 7
class StripeClient
{
... lines 10 - 40
    public function createInvoiceItem($amount, User $user, $description)
    {
... lines 43 - 48
    }
... lines 50 - 63
}
```

Copy that code from our controller, remove it, and paste it here. Update amount to use $amount and description to use $description. Add a return statement just in case:

```
65 lines | src/AppBundle/StripeClient.php
... lines 1 - 7
class StripeClient
{
... lines 10 - 40
    public function createInvoiceItem($amount, User $user, $description)
    {
        return \Stripe\InvoiceItem::create(array(
            "amount" => $amount,
            "currency" => "usd",
            "customer" => $user->getStripeCustomerId(),
            "description" => $description
        ));
    }
... lines 50 - 63
}
```

In OrderController, call this $stripeClient->createInvoiceItem() passing it $product->getPrice() * 100, $user and $product->getName():

```
70 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 47
            foreach ($this->get('shopping_cart')->getProducts() as $product) {
                $stripeClient->createInvoiceItem(
                    $product->getPrice() * 100,
                    $user,
                    $product->getName()
                );
            }
... lines 55 - 60
        }
... lines 62 - 68
    }
}
```

Perfect! For the last piece, add a new public function createInvoice() with a $user whose customer we should invoice and a $payImmediately argument that defaults to true:

```
65 lines | src/AppBundle/StripeClient.php
... lines 1 - 7
class StripeClient
{
... lines 10 - 50
    public function createInvoice(User $user, $payImmediately = true)
    {
... lines 53 - 62
    }
}
```

Who knows, there might be some time in the future when we *don't* want to pay an invoice immediately.

You know the drill: copy the invoice code from the controller, remove it and paste it into StripeClient. Wrap the pay() method inside if ($payImmediately). Finally, return the $invoice:

65 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 7
class StripeClient
{
... lines 10 - 50
    public function createInvoice(User $user, $payImmediately = true)
    {
        $invoice = \Stripe\Invoice::create(array(
            "customer" => $user->getStripeCustomerId()
        ));

        if ($payImmediately) {
            // guarantee it charges *right* now
            $invoice->pay();
        }

        return $invoice;
    }
}
```

Call that in the controller: $stripeClient->createInvoice() passing it $user and true to pay immediately:

70 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 54
            $stripeClient->createInvoice($user, true);
... lines 56 - 60
        }
... lines 62 - 68
    }
}
```

Phew! This was a giant step sideways - but not only is our code more re-usable, it just makes a lot more sense when you read it!

Double-check to make sure it works. Add something to your cart. Check-out. Yes! No error! The system still works and this StripeClient is really, really sweet.

Chapter 9: Force HTTPS ... please

Guess what? You could deploy this code *right* now. Sure - we have a lot more to talk about - like subscriptions & discounts - but the system is ready.

Oh, but there's just one thing that you *cannot* forget to do. And that's to *force* https to be used on your checkout page.

Right now, there is *no* little lock icon in my browser - this page is *not* secure. Of course it's not a problem now because I'm just coding locally.

But on production, different story. Even though you're not handling credit card information, Stripe *does* submit that token to our server. If that submit happens over a non-https connection, that's a security risk: there *could* be somebody in the middle reading that token. Regardless of what they might or might not be able to do with that, we need to avoid this.

There are *a lot* of ways to force HTTPs, but let me show you my *favorite* in Symfony. In OrderController, right above checkoutAction(), this @Route annotation is what defines the URL to this page. At the end of this, add a new option called schemes set to two curly braces and a set of double-quotes with https inside:

70 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 27
    /**
     * @Route("/checkout", name="order_checkout", schemes={"https"})
... line 30
     */
    public function checkoutAction(Request $request)
    {
... lines 34 - 68
    }
}
```

OK, go back and refresh! Cool! Symfony automatically redirects me to https. Life is good.

No HTTPS in dev Please!

Wait, life is *not* good. I *hate* needing to setup SSL certificates on my local machine. I actually have one setup already, but other developers might not. That's a huge pain for them... for no benefit.

Fortunately, there's a trick. Replace https, with %secure_channel%:

```
71 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 27
    /**
     * @Route("/checkout", name="order_checkout", schemes={"%secure_channel%"})
... line 30
     */
    public function checkoutAction(Request $request)
    {
... lines 34 - 68
    }
}
```

This syntax is referencing a *parameter* in Symfony, so basically a configuration variable. Open parameters.yml, add a new secure_channel parameter and set it to http:

```
25 lines | app/config/parameters.yml.dist
... lines 1 - 3
parameters:
... lines 5 - 22
    # set to https on production
    secure_channel: http
```

And as you know, if you add a key here, also add it to parameters.yml.dist:

```
25 lines | app/config/parameters.yml.dist
... lines 1 - 3
parameters:
... lines 5 - 22
    # set to https on production
    secure_channel: http
```

Ok, head back to the homepage: http://localhost:8000 and click to checkout. Hey! We're back in http. When you deploy, change that setting to https and *boom*, your checkout will be secure.

So there's your little trick for forcing https without being forced to hate your life while coding.

Chapter 10: Building the Custom Checkout Form

Earlier, we were *rushing* to get the site up and the sheep shopping. That's why we used Stripe's pre-built embedded form. And this is *completely* fine if you like it. But I want to build a custom form that looks like native on our site.

To do that, go back to the Stripe docs. Instead of embedded form, click "Custom Form". Using a custom form is *very* similar: we still send the credit card information to Stripe, and Stripe will still give us back a token. The difference is that *we* are responsible for building the HTML form.

Setting up the Stripe JavaScript

To help communicate with Stripe, we need some JavaScript. Copy the first JavaScript code and then find the checkout.html.twig template. At the top, override {% block javascripts %} and then call the {{ parent() }} function. Paste the script tag below:

52 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block javascripts %}
    {{ parent() }}

    <script type="text/javascript" src="https://js.stripe.com/v2/"></script>
... lines 8 - 11
{% endblock %}
... lines 13 - 52
```

This is just the Twig way of adding some new JavaScript to our page. The base layout *also* has a javascripts block and jQuery is already included:

83 lines | app/Resources/views/base.html.twig

```
... line 1
<html>
... lines 3 - 14
    <body>
... lines 16 - 73
        {% block javascripts %}
            <script src="https://code.jquery.com/jquery-3.1.0.js"
                    integrity="sha256-slogkvB1K3VOkzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="
                    crossorigin="anonymous"></script>
... lines 78 - 79
        {% endblock %}
    </body>
</html>
```

Next, we need to tell the JavaScript about our publishable key. Copy that code from the docs and add it in the block:

```
52 lines | app/Resources/views/order/checkout.html.twig
    ... lines 1 - 3
{% block javascripts %}
    {{ parent() }}

    <script type="text/javascript" src="https://js.stripe.com/v2/"></script>

    <script type="text/javascript">
        Stripe.setPublishableKey('{{ stripe_public_key }}');
    </script>
{% endblock %}
    ... lines 13 - 52
```

We already know from our original code that we have a variable called stripe_public_key. Inside of the JavaScript quotes, print stripe_public_key:

```
52 lines | app/Resources/views/order/checkout.html.twig
    ... lines 1 - 3
{% block javascripts %}
    ... lines 5 - 8
    <script type="text/javascript">
        Stripe.setPublishableKey('{{ stripe_public_key }}');
    </script>
{% endblock %}
    ... lines 13 - 52
```

Awesome!

Rendering the HTML Form

With that done, it's time to build the form itself. And surprise! I already built us a basic HTML form. Delete the old, embedded form code. Replace it with {{ include('order/_cardForm.html.twig') }}:

```
52 lines | app/Resources/views/order/checkout.html.twig
    ... lines 1 - 13
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
    ... lines 18 - 44
            <div class="col-xs-12 col-sm-6">
                {{ include('order/_cardForm.html.twig') }}
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

This will read this *other* template file I prepared: _cardForm.html.twig:

```
68 lines | app/Resources/views/order/_cardForm.html.twig
<form action="" method="POST" class="js-checkout-form checkout-form">
    <div class="row">
        <div class="col-xs-8 col-sm-6 col-sm-offset-2 form-group">
            <div class="input-group">
                <span class="input-group-addon">
                    <i class="fa fa-user"></i>
```

```html
         <i class="fa fa-user"></i>
       </span>
      <input data-stripe="name" class="form-control" type="text" autocomplete="off" id="card-name" required placeholder="Card H
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-8 col-sm-6 col-sm-offset-2 form-group">
    <div class="input-group">
      <span class="input-group-addon">
         <i class="fa fa-credit-card"></i>
       </span>
      <input data-stripe="number" type="text" autocomplete="off" class="form-control js-cc-number" id="card-number" required pl
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-4 col-sm-3 col-sm-offset-2 form-group">
    <div class="input-group">
      <span class="input-group-addon">
         <i class="fa fa-calendar-o"></i>
       </span>
      <input data-stripe="exp" type="text" size="4" autocomplete="off" class="form-control js-cc-exp" id="card-expiration" required
    </div>
  </div>
  <div class="col-xs-4 col-sm-3 form-group">
    <div class="input-group">
      <span class="input-group-addon">
         <i class="fa fa-lock"></i>
       </span>
      <input data-stripe="cvc" type="text" size="4" autocomplete="off" class="form-control js-cc-cvc" id="card-cvc" required="requi
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-8 col-sm-3 col-sm-offset-2 form-group">
    <div class="input-group">
      <span class="input-group-addon">
         <i class="fa fa-map-marker"></i>
       </span>
      <input type="text" autocomplete="off" class="form-control" id="card-zip" placeholder="Zip"/>
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-8 col-sm-6 col-sm-offset-2 text-center">
    <div class="alert alert-danger js-checkout-error hidden"></div>
  </div>
</div>

<div class="row">
  <div class="col-xs-8 col-sm-6 col-sm-offset-2 text-center">
```

```
            <button type="submit" class="js-submit-button btn btn-lg btn-danger">
                Checkout
            </button>
        </div>
    </div>
</form>
```

As you can see, this is a normal HTML form. Its method is post and its action is still empty so that it will submit right back to the same URL and controller. Then, there's just a bunch of fields that are rendered to look good with Bootstrap.

Let's see how awesome my design skills are: go back and refresh. Hey, it looks pretty good! Probably because someone styled this for me.

Do NOT Submit Card Data to your Server

There are a few *really* important things about this form. Most importantly, notice that the input fields have *no* name attribute. This is crucial. Eventually, we *will* submit this form, but we do *not* want to submit these fields because we do *not* want credit card information passing through our server. Because these fields do *not* have a name attribute, they are *not* submitted.

So instead of name, Stripe asks you to use a data-stripe attribute. This tells Stripe *which* data this field holds. Since this is the cardholder name, we have data-stripe="name". Then below, data-stripe="number", data-stripe="exp" and so-on.

But I'm not choosing these values at random. Inside Stripe's documentation, it tells you which data-stripe value to use for each piece. If you follow the rules, Stripe's JavaScript will do all the work of collecting this data and sending it to Stripe.

OK, let's hook up that JavaScript logic next.

Chapter 11: Checkout Form JS Handling Logic

Now that we've got the form in place, we need to add some JavaScript that will send all the card information to Stripe. Once again, Stripe wrote a lot of this code for us. Over-achiever.

Copy their JavaScript and scroll up and paste it with our other JavaScript:

```
89 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
        Stripe.setPublishableKey('{{ stripe_public_key }}');

        $(function () {
            var $form = $('#payment-form');
            $form.submit(function (event) {
                // Disable the submit button to prevent repeated clicks:
                $form.find('.submit').prop('disabled', true);

                // Request a token from Stripe:
                Stripe.card.createToken($form, stripeResponseHandler);

                // Prevent the form from being submitted:
                return false;
            });
        });
... lines 25 - 47
    </script>
{% endblock %}
... lines 50 - 89
```

Then, back on the docs, scroll down a little further to another function called stripeResponseHandler(). Copy that too and paste it:

89 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
        Stripe.setPublishableKey('{{ stripe_public_key }}');

        $(function () {
            var $form = $('#payment-form');
            $form.submit(function (event) {
                // Disable the submit button to prevent repeated clicks:
                $form.find('.submit').prop('disabled', true);

                // Request a token from Stripe:
                Stripe.card.createToken($form, stripeResponseHandler);

                // Prevent the form from being submitted:
                return false;
            });
        });

        function stripeResponseHandler(status, response) {
            // Grab the form:
            var $form = $('#payment-form');

            if (response.error) { // Problem!

                // Show the errors on the form:
                $form.find('.payment-errors').text(response.error.message);
                $form.find('.submit').prop('disabled', false); // Re-enable submission

            } else { // Token was created!

                // Get the token ID:
                var token = response.id;

                // Insert the token ID into the form so it gets submitted to the server:
                $form.append($('<input type="hidden" name="stripeToken">').val(token));

                // Submit the form:
                $form.get(0).submit();
            }
        }
    </script>
{% endblock %}
... lines 50 - 89
```

Prepping the JS

Let's look at the code: it uses a jQuery document.ready block to find the form and attach an on submit handler function:

```
89 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 11
        $(function () {
            var $form = $('#payment-form');
            $form.submit(function (event) {
... lines 15 - 22
            });
        });
... lines 25 - 47
    </script>
{% endblock %}
... lines 50 - 89
```

Because basically, when the user submits the form, we *don't* want to submit the form! Ahem, I mean, we want to stop, send all the information to Stripe, wait for the token to come back, put *that* in the form, and *then* submit it.

In our case, I've given the form a class called js-checkout-form:

```
68 lines | app/Resources/views/order/_cardForm.html.twig
<form action="" method="POST" class="js-checkout-form checkout-form">
... lines 2 - 66
</form>
```

Copy that class, and change the JavaScript to look for .js-checkout-form:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 11
        $(function () {
            var $form = $('.js-checkout-form');
... lines 14 - 22
        });
... lines 24 - 50
    </script>
{% endblock %}
... lines 53 - 92
```

This is referenced in *one* other spot further below. Update that too:

```
92 lines | app/Resources/views/order/checkout.html.twig
 ... lines 1 - 3
{% block javascripts %}
 ... lines 5 - 8
    <script type="text/javascript">
 ... lines 10 - 24
        function stripeResponseHandler(status, response) {
            // Grab the form:
            var $form = $('.js-checkout-form');
 ... lines 28 - 49
        }
    </script>
{% endblock %}
 ... lines 53 - 92
```

It's not the most organized JS code.

Oh, and you'll notice that I use these js- classes a lot in my html. That's a standard that *I* like to use whenever I give an element a class *not* because I want to style it, but because I want to find it with JavaScript.

When this form is submitted, add event.preventDefault() to prevent the form from *actually* submitting:

```
92 lines | app/Resources/views/order/checkout.html.twig
 ... lines 1 - 3
{% block javascripts %}
 ... lines 5 - 8
    <script type="text/javascript">
 ... lines 10 - 11
        $(function () {
 ... line 13
            $form.submit(function (event) {
                event.preventDefault();
 ... lines 16 - 21
            });
        });
 ... lines 24 - 50
    </script>
{% endblock %}
 ... lines 53 - 92
```

This does more-or-less the same thing as returning false at the end of the function, but with some subtle differences.

Oof - let me fix *some* of this bad indentation. Next, the code finds the submit button so it can disable it. In our form, the button has a js-submit-button class:

```
68 lines | app/Resources/views/order/_cardForm.html.twig
<form action="" method="POST" class="js-checkout-form checkout-form">
 ... lines 2 - 59
    <div class="row">
        <div class="col-xs-8 col-sm-6 col-sm-offset-2 text-center">
            <button type="submit" class="js-submit-button btn btn-lg btn-danger">
                Checkout
            </button>
        </div>
    </div>
</form>
```

Copy that and update the code here, and once more down below:

92 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 11
        $(function () {
... line 13
            $form.submit(function (event) {
... lines 15 - 16
                // Disable the submit button to prevent repeated clicks:
                $form.find('.js-submit-button').prop('disabled', true);
... lines 19 - 21
            });
        });
... line 24
        function stripeResponseHandler(status, response) {
... lines 26 - 28
            if (response.error) { // Problem!
... lines 30 - 34
                $form.find('.js-submit-button').prop('disabled', false); // Re-enable submission
... lines 36 - 48
            }
        }
    </script>
{% endblock %}
... lines 53 - 92
```

Fetching and Using the Token

Finally, here is the *meat* of the code. When we call Stripe.card.createToken():

92 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 11
        $(function () {
... line 13
            $form.submit(function (event) {
... lines 15 - 19
                // Request a token from Stripe:
                Stripe.card.createToken($form, stripeResponseHandler);
            });
        });
... lines 24 - 50
    </script>
{% endblock %}
... lines 53 - 92
```

Stripe's Javascript will automatically fetch all the credit card data by reading the data-stripe attributes. Then, it sends those to stripe via AJAX. When that call finishes, it will execute the stripeResponseHandler function, and hopefully the response will contain that all-important token:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 24
        function stripeResponseHandler(status, response) {
... lines 26 - 49
        }
    </script>
{% endblock %}
... lines 53 - 92
```

Now, if there was a problem with that card - like an invalid expiration - we need to show that error to the user. To do that, it looks for a payment-errors class and puts the message inside of that:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 24
        function stripeResponseHandler(status, response) {
... lines 26 - 28
            if (response.error) { // Problem!

                // Show the errors on the form:
                $form.find('.js-checkout-error')
                    .text(response.error.message)
... lines 34 - 48
            }
        }
    </script>
{% endblock %}
... lines 53 - 92
```

I have a div ready for this. Its class is js-checkout-error and its hidden by default:

```
68 lines | app/Resources/views/order/_cardForm.html.twig
<form action="" method="POST" class="js-checkout-form checkout-form">
... lines 2 - 53
    <div class="row">
        <div class="col-xs-8 col-sm-6 col-sm-offset-2 text-center">
            <div class="alert alert-danger js-checkout-error hidden"></div>
        </div>
    </div>
... lines 59 - 66
</form>
```

Change the selector to .js-checkout-error, set the text, but then *also* call removeClass('hidden') so the element shows up:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 24
        function stripeResponseHandler(status, response) {
... lines 26 - 28
            if (response.error) { // Problem!

                // Show the errors on the form:
                $form.find('.js-checkout-error')
                    .text(response.error.message)
                    .removeClass('hidden');
... lines 35 - 48
            }
        }
    </script>
{% endblock %}
... lines 53 - 92
```

Below in the else, life is good!! I'll paste the .js-checkout-error code from before and modify it to re-add the hidden class - since now things are successful:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script type="text/javascript">
... lines 10 - 24
        function stripeResponseHandler(status, response) {
... lines 26 - 28
            if (response.error) { // Problem!
... lines 30 - 36
            } else { // Token was created!
                $form.find('.js-checkout-error')
                    .addClass('hidden');
... lines 40 - 48
            }
        }
    </script>
{% endblock %}
... lines 53 - 92
```

When things work, the response comes back with a token, which we get via response.id:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
  <script type="text/javascript">
... lines 10 - 24
    function stripeResponseHandler(status, response) {
... lines 26 - 28
      if (response.error) { // Problem!
... lines 30 - 36
      } else { // Token was created!
... lines 38 - 40
        // Get the token ID:
        var token = response.id;
... lines 43 - 48
      }
    }
  </script>
{% endblock %}
... lines 53 - 92
```

To send this to the server, we just smash it into a new input hidden field called... drumroll ... stripeToken:

```
92 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
  <script type="text/javascript">
... lines 10 - 24
    function stripeResponseHandler(status, response) {
... lines 26 - 28
      if (response.error) { // Problem!
... lines 30 - 36
      } else { // Token was created!
... lines 38 - 43
        // Insert the token ID into the form so it gets submitted to the server:
        $form.append($('<input type="hidden" name="stripeToken">').val(token));
... lines 46 - 48
      }
    }
  </script>
{% endblock %}
... lines 53 - 92
```

This is *precisely* what the embedded form did. Once the form is submitted, the controller will hum along like nothing ever changed.

Testing the Error and Success

But, that's assuming we didn't mess something up! That's a big but. Go back and refresh the page.

First, test that the error handling works by adding an expiration date in the *past*. Put in the real credit card number -- oof, ugly formatting - we'll fix that. Then, use an expired expiration. Hit checkout and... boom!

It sent the info to Stripe, Stripe came back with an error, we put the error in the box, and showed that box to our user. In other words, we're awesome. Change this to a *future* expiration and try again.

It's alive!!!

The only problem I can think of now is how *ugly* entering a credit card number is: all those numbers just run together. The expiration field is a mess too. Oof. Let's fix that - it's surprisingly easy!

Chapter 12: Pretty Card Formatting

The *old*, embedded form had a couple of nice formatting behaviors - like automatically adding a space between every 4 card numbers. Fortunately, Stripe has us covered once again here. Go back to the documentation and scroll down - they eventually reference something called jQuery.payment: a neat little JavaScript library for formatting checkout fields nicely.

It even provides validation, in case you want to make sure the numbers are sane before sending them off to Stripe.

I've already downloaded this library into the web/js directory, so all we need to do is include it on the page and point it at our form.

At the top, add a new script tag and set its src="js/jQuery.payment.min.js":

98 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 6
    <script type="text/javascript" src="https://js.stripe.com/v2/"></script>
    <script src="{{ asset('js/jquery.payment.min.js') }}"></script>
... lines 9 - 57
{% endblock %}
... lines 59 - 98
```

The asset function is an optional helper function from Symfony - nothing magic going on there.

Then, down below... try to ignore the ugly indentation that I should have fixed earlier, and say $form.find(). We need to find the credit card number input. But don't worry! I planned ahead and gave it a special js-cc-number class. I also added js-cc-exp and js-cc-cvc:

68 lines | app/Resources/views/order/_cardForm.html.twig

```
<form action="" method="POST" class="js-checkout-form checkout-form">
... lines 2 - 12
  <div class="row">
    <div class="col-xs-8 col-sm-6 col-sm-offset-2 form-group">
      <div class="input-group">
... lines 16 - 18
        <input data-stripe="number" type="text" autocomplete="off" class="form-control js-cc-number" id="card-number" required pl
      </div>
    </div>
  </div>

  <div class="row">
    <div class="col-xs-4 col-sm-3 col-sm-offset-2 form-group">
      <div class="input-group">
... lines 27 - 29
        <input data-stripe="exp" type="text" size="4" autocomplete="off" class="form-control js-cc-exp" id="card-expiration" required
      </div>
    </div>
    <div class="col-xs-4 col-sm-3 form-group">
      <div class="input-group">
... lines 35 - 37
        <input data-stripe="cvc" type="text" size="4" autocomplete="off" class="form-control js-cc-cvc" id="card-cvc" required="requi
      </div>
    </div>
  </div>
... lines 42 - 66
</form>
```

Fill in .js-cc-number and then call .payment('formatCardNumber'):

98 lines | app/Resources/views/order/checkout.html.twig

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 9
  <script type="text/javascript">
... lines 11 - 12
    $(function () {
      var $form = $('.js-checkout-form');

      $form.find('.js-cc-number').payment('formatCardNumber');
... lines 17 - 28
    });
... lines 30 - 56
  </script>
{% endblock %}
... lines 59 - 98
```

Repeat this two more times for js-cc-exp with formatCardExpiry and formatCardCVC. Don't forget to update that class name too:

```
98 lines | app/Resources/views/order/checkout.html.twig
... lines 1 - 15
            $form.find('.js-cc-number').payment('formatCardNumber');
            $form.find('.js-cc-exp').payment('formatCardExpiry');
            $form.find('.js-cc-cvc').payment('formatCardCVC');
... lines 19 - 98
```

Try it out! So sweet! The card field gets pretty auto-spacing and even *more* importantly, the library adds the slash automatically for the expiration field. It also limits the CVC field to a maximum of 4 numbers.

So custom forms are a little bit more work. But they fundamentally work the same.

Before we finish, there's one big hole left in our setup: failing gracefully when someone's card is declined.

Chapter 13: Pro Error Handling

A lot of failures are stopped right here: instead of passing us back a token, Stripe tells us something is wrong and we tell the user immediately.

But guess what? We are not handling all cases where things can go wrong. Go back to the Stripe documentation and click the link called "Testing". This page is full of helpful information about how to *test* your Stripe setup. One of the most interesting parts is this cool table of fake credit card numbers that you can use in the Test environment. They include cards that will work, but also cards that will *fail*, for various reasons.

Ah, this one is particularly important: this number will look valid, but will be declined when we try to charge it.

Let's try this out. Use: 4000 0000 0000 0002. Give it a valid expiration and then hit enter. It's submitting, but woh! A huge 500 error:

> Your card was declined.

This is a problem: on production, this would be a big error screen with no information. Instead, we need to be able to tell our user what went wrong: we need to be able to say "Hey Buddy! You card was declined".

Stripe Error Handling 101

Let's talk about how Stripe handles errors. First, on the error page, if I hover over this little word, it tells me that a Stripe\Error\Card exception object was thrown. Whenever you make an API request to Stripe, it will either be successful, or Stripe will throw an exception.

On Stripe's API documentation, near the top, they have a section that talks about Errors.

There are a few important things. First, Stripe uses different status codes to give you some information about what went wrong. That's cool, but these *types* are more important. When you make an API request to Stripe and it fails, Stripe will send back a JSON response with a type key. That type will be one of these values. This goes a *long* way to telling you what went wrong.

So, how can we read the type inside our code?

Open up the vendor/stripe directory to look at the SDK code. Hey, check this out: the library has a custom Exception class for *each* of the possible type values. For example, if type is card_error, the library throws a Card exception, which is what we're getting right now.

But if Stripe was rate limiting us because we made way too many requests, Stripe would throw a RateLimit exception. This means that we can use a try-catch block to handle *only* specific error types.

Isolating the Checkout Code

The *one* error we need to handle is card_error - because this happens when a card is declined.

To do that, let's move all of this processing logic into its own private function in this class. That'll make things cleaner.

To do this, I'll use a PhpStorm shortcut: select the code, hit Control+T (or go to the "Refactor"->"Refactor This" menu) and select "Method". Create a new method called chargeCustomer(). Hit refactor:

80 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        if ($request->isMethod('POST')) {
... lines 37 - 38
            $this->chargeCustomer($token);
... lines 40 - 44
        }
... lines 46 - 52
    }

    /**
     * @param $token
     */
    private function chargeCustomer($token)
    {
        $stripeClient = $this->get('stripe_client');
        /** @var User $user */
        $user = $this->getUser();
        if (!$user->getStripeCustomerId()) {
            $stripeClient->createCustomer($user, $token);
        } else {
            $stripeClient->updateCustomerCard($user, $token);
        }

        foreach ($this->get('shopping_cart')->getProducts() as $product) {
            $stripeClient->createInvoiceItem(
                $product->getPrice() * 100,
                $user,
                $product->getName()
            );
        }
        $stripeClient->createInvoice($user, true);
    }
}
... lines 79 - 80
```

You don't need PhpStorm to do that: it just moved my code down into this private function and called that function from the original spot.

Handling the Card Exception

OK, back to business: we know that when a card is declined, something in that code will throw a Stripe\Error\Card exception. I'm adding a little documentation just to indicate this:

```
89 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 62
    /**
     * @param $token
     * @throws \Stripe\Error\Card
     */
    private function chargeCustomer($token)
    {
... lines 69 - 85
    }
}
... lines 88 - 89
```

Back in checkoutAction(), add a new $error = false variable before the if, because at this point, no error has occurred:

```
89 lines | src/AppBundle/Controller/OrderController.php
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        $error = false;
        if ($request->isMethod('POST')) {
... lines 38 - 51
        }
... lines 53 - 60
    }
... lines 62 - 86
}
... lines 88 - 89
```

Next, surround the chargeCustomer() call in a try-catch: try chargeCustomer() and then catch *just* a \Stripe\Error\Card exception:

89 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        $error = false;
        if ($request->isMethod('POST')) {
... lines 38 - 39
            try {
                $this->chargeCustomer($token);
            } catch (\Stripe\Error\Card $e) {
... line 43
            }
... lines 45 - 51
        }
... lines 53 - 60
    }
... lines 62 - 86
}
... lines 88 - 89
```

If we get here, there was a problem charging the card. Update $error to some nice message, like: "There was a problem charging your card.". Then add $e->getMessage():

89 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 35
        $error = false;
        if ($request->isMethod('POST')) {
... lines 38 - 39
            try {
                $this->chargeCustomer($token);
            } catch (\Stripe\Error\Card $e) {
                $error = 'There was a problem charging your card: '.$e->getMessage();
            }
... lines 45 - 51
        }
... lines 53 - 60
    }
... lines 62 - 86
}
... lines 88 - 89
```

That's will be the message that Stripe's sending back like, "Your card was declined," or, "Your card cannot be used for this type of transaction."

Now, if there *is* an error, we don't want to empty the cart, we don't want to add the nice message and we don't want to redirect. So, if (!$error), then it's safe to do those things:

89 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 35
        $error = false;
        if ($request->isMethod('POST')) {
... lines 38 - 39
            try {
                $this->chargeCustomer($token);
            } catch (\Stripe\Error\Card $e) {
                $error = 'There was a problem charging your card: '.$e->getMessage();
            }

            if (!$error) {
                $this->get('shopping_cart')->emptyCart();
                $this->addFlash('success', 'Order Complete! Yay!');

                return $this->redirectToRoute('homepage');
            }
        }
... lines 53 - 60
    }
... lines 62 - 86
}
... lines 88 - 89
```

If there *is* an error, our code will continue down and it will re-render the checkout template, which is exactly what we want! Pass in the $error variable so we can show it to the user:

89 lines | src/AppBundle/Controller/OrderController.php

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 31
    public function checkoutAction(Request $request)
    {
... lines 34 - 53
        return $this->render('order/checkout.html.twig', array(
... lines 55 - 57
            'error' => $error,
        ));

    }
... lines 62 - 86
}
... lines 88 - 89
```

Then, in then template, specifically the _cardForm template, render error inside of our error div:

68 lines | app/Resources/views/order/_cardForm.html.twig

```
<form action="" method="POST" class="js-checkout-form checkout-form">
... lines 2 - 53
    <div class="row">
        <div class="col-xs-8 col-sm-6 col-sm-offset-2 text-center">
            <div class="alert alert-danger js-checkout-error {{ error ? '' : 'hidden' }}">{{ error }}</div>
        </div>
    </div>
... lines 59 - 66
</form>
```

If there is no error, no problem! It won't render anything. If there *is* an error, then we we need to *not* render the hidden class. Use an inline if statement to say:

> If error, then don't render any class, else render the hidden class

A little tricky, but that should do it.

Ok, let's try it again. Refresh the page. Put in the fake credit card: the number 4, a thousand zeroes, and a 2 at the end. Finish it up and submit.

There's the error! Setup, complete.

Ok, guys, you have a killer checkout system via Stripe. In part 2 of this course, we're going to talk about where things get a little bit more difficult: like subscriptions and discounts. This includes handling web hooks, one of the scariest and toughest parts of subscriptions.

But, don't stop - go make a *great* product and sell it.

All right, guys, seeya next time.