Stripe Level 2: Subscriptions, Discounts, Webhooks, oh my!

With <3 from SymfonyCasts

Chapter 1: Create those Subscription Plans

Yes! You're back! You survived Stripe part 1. Ok... that's cool, but now we have bigger fish to fry. So fasten your seat belts, put on a pot of coffee, and get ready to sharpen your Stripe subscription skills.

Wake up the Sheep

To become my best friend, just code along with me! If you coded with part one of the tutorial, you'll need a fresh copy of the code: we made a few tweaks. Download it from this page, unzip it, and then move into the start/ directory.

That'll give you the same code I have here. Open the README file for the setup instructions. The last step will be to open a terminal, move into the project directory, and run:

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

to start the built-in web server. Now for the magic: pull up the site at http://localhost:8000.

Oh yea, it's the Sheep Shear Club: thanks to part 1, it's already possible to buy individual products. Now, click "Pricing". The *real* point of the site is to get a *subscription* that'll send your awesome sheep awesome shearing accessories monthly. After vigorous meetings with our investors, we've decided to offer 2 plans: the Farmer Brent at $99/month and the New Zealander at $199/month. But other than this fancy looking page, we haven't coded anything for this yet.

Tell Stripe about the Plans

Our first step is to tell Stripe about these two plans. To do that, open your trusty Stripe dashboard, make sure you're in the test environment, and find "Plans" on the left. We're going to create the two plans in Stripe by hand. The ID can be any unique string and we'll use this forever after to refer to the plan. Use, farmer_brent_monthly. Then, fill in a name - that's less important and set the price as $99 per month. Create that plan!

And yes, later, we'll add *yearly* versions of each plan... but one step at a time! Repeat that whole process for the New Zealander: set its ID to new_zealander_monthly, give it a name and set its price and interval. Perfect!

Manage the Plans in Our Code

Now, head back to our code and open the src/AppBundle/Subscription directory. I added two classes to help us stay organized. The first is called SubscriptionHelper:

33 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 2
namespace AppBundle\Subscription;

class SubscriptionHelper
{
    /** @var SubscriptionPlan[] */
    private $plans = [];

    public function __construct()
    {
        // todo - add the plans
//        $this->plans[] = new SubscriptionPlan(
//            'STRIPE_PLAN_KEY',
//            'OUR PLAN NAME',
//            'PRICE'
//        );
    }

    /**
     * @param $planId
     * @return SubscriptionPlan|null
     */
    public function findPlan($planId)
    {
        foreach ($this->plans as $plan) {
            if ($plan->getPlanId() == $planId) {
                return $plan;
            }
        }
    }
}
```

And as you can see... it's not very helpful... yet. But pretty soon, it will keep track of the two plans we just added. We'll use the second class - SubscriptionPlan:

34 lines | src/AppBundle/Subscription/SubscriptionPlan.php

```
... lines 1 - 2
namespace AppBundle\Subscription;

class SubscriptionPlan
{
    private $planId;

    private $name;

    private $price;

    public function __construct($planId, $name, $price)
    {
        $this->planId = $planId;
        $this->name = $name;
        $this->price = $price;
    }

    public function getPlanId()
    {
        return $this->planId;
    }

    public function getName()
    {
        return $this->name;
    }

    public function getPrice()
    {
        return $this->price;
    }
}
```

To hold the data for each plan: the plan ID, name and price. But we won't save these to the database.

In SubscriptionHelper, setup the two plans. For the first, use farmer_brent_monthly for the key, name it Farmer Brent and use 99 as the price:

38 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 4
class SubscriptionHelper
{
... lines 7 - 9
    public function __construct()
    {
        $this->plans[] = new SubscriptionPlan(
            'farmer_brent_monthly',
            'Farmer Brent',
            99
        );
... lines 17 - 22
    }
... lines 24 - 36
}
```

Copy that and repeat the same thing: new_zealander_monthly, New Zealander and 199:

38 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 4
class SubscriptionHelper
{
... lines 7 - 9
    public function __construct()
    {
... lines 12 - 17
        $this->plans[] = new SubscriptionPlan(
            'new_zealander_monthly',
            'New Zealander',
            199
        );
    }
... lines 24 - 36
}
```

Love it! The 2 plans live in Stripe *and* in our code. Time to make it possible to add these to our cart, and then checkout.

Chapter 2: Add the Subscription to your Cart

We can already add products to our cart... but a user should *also* be able to click these fancy buttons and add a subscription to their cart.

Open up OrderController: the home for the checkout and shopping cart magic. I've already started a new page called addSubscriptionToCartAction():

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 27
    /**
     * @Route("/cart/subscription/{planId}", name="order_add_subscription_to_cart")
     */
    public function addSubscriptionToCartAction($planId)
    {
        // todo - add the subscription plan to the cart!
    }
... lines 35 - 96
}
... lines 98 - 99
```

When we're done, if the user goes to /cart/subscription/farmer_brent_monthly, this should put that into the cart.

First, hook up the buttons to point here. The template for this page lives at app/Resources/views/product/pricing.html.twig:

106 lines | app/Resources/views/product/pricing.html.twig

```
... lines 1 - 3
{% block body %}

<div class="container">
... lines 7 - 10
    <div class="row">
        <div class="col-md-6">
            <div class="price-square">
                <p class="pricing-length-header monthly text-center">Farmer Brent</p>
                <div class="pricing-info-padding">
                    <p class="price text-center">$99</p>
                    <p class="text-center">A fresh set of shears monthly along with our beloved after-shear in one of two fragrances!</p>
                </div>
                <a href="{{ path('order_add_subscription_to_cart', {
                    'planId': 'TODO'
                }) }}" class="btn btn-warning center-block price-btn">
                    Shear Me!
                </a>
            </div>
        </div>

        <div class="col-md-6">
            <div class="price-square price-square-yearly">
                <p class="pricing-length-header yearly text-center">New Zealander</p>
                <div class="pricing-info-padding">
                    <p class="price text-center">$199</p>
                    <p class="text-center">The perks of a Farmer Brent membership, but with a fresh bottle of conditioner and a comb each
                </div>
                <a href="{{ path('order_add_subscription_to_cart', {
                    'planId': 'TODO'
                }) }}" class="btn btn-warning center-block price-btn">
                    Get Shearing!
                </a>
            </div>
        </div>

    </div>
... lines 43 - 102
</div>

{% endblock %}
```

I started adding the link to this page, but left the plan ID blank. Fill 'em in! farmer_brent_monthly and then down below, new_zealander_monthly:

106 lines | app/Resources/views/product/pricing.html.twig

```
... lines 1 - 3
{% block body %}

<div class="container">
... lines 7 - 10
    <div class="row">
        <div class="col-md-6">
            <div class="price-square">
                <p class="pricing-length-header monthly text-center">Farmer Brent</p>
                <div class="pricing-info-padding">
                    <p class="price text-center">$99</p>
                    <p class="text-center">A fresh set of shears monthly along with our beloved after-shear in one of two fragrances!</p>
                </div>
                <a href="{{ path('order_add_subscription_to_cart', {
                    'planId': 'farmer_brent_monthly'
                }) }}" class="btn btn-warning center-block price-btn">
                    Shear Me!
                </a>
            </div>
        </div>

        <div class="col-md-6">
            <div class="price-square price-square-yearly">
                <p class="pricing-length-header yearly text-center">New Zealander</p>
                <div class="pricing-info-padding">
                    <p class="price text-center">$199</p>
                    <p class="text-center">The perks of a Farmer Brent membership, but with a fresh bottle of conditioner and a comb each
                </div>
                <a href="{{ path('order_add_subscription_to_cart', {
                    'planId': 'new_zealander_monthly'
                }) }}" class="btn btn-warning center-block price-btn">
                    Get Shearing!
                </a>
            </div>
        </div>

    </div>
... lines 43 - 102
</div>

{% endblock %}
```

Go back to that page and refresh. The links look great!

Put the Plan in the Cart

Now back to the controller! In the first Stripe tutorial, we worked with a ShoppingCart class that I created for us... because it's not really that important. It basically allows you to store products and a subscription in the user's session, so that as they surf around, we know what they have in their cart.

But before we use that, first get an instance of our new SubscriptionHelper object with $subscriptionHelper = $this->get('subscription_helper'):

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 30
31      public function addSubscriptionToCartAction($planId)
32      {
33          $subscriptionHelper = $this->get('subscription_helper');
... lines 34 - 42
43      }
... lines 44 - 105
106 }
... lines 107 - 108
```

I already registered this as a service in Symfony:

19 lines | app/config/services.yml

```
... lines 1 - 5
6   services:
... lines 7 - 15
16      subscription_helper:
17          class: AppBundle\Subscription\SubscriptionHelper
18          autowire: true
```

Next, add $plan = $subscriptionHelper->findPlan() and pass it the $planId:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 30
31      public function addSubscriptionToCartAction($planId)
32      {
33          $subscriptionHelper = $this->get('subscription_helper');
34          $plan = $subscriptionHelper->findPlan($planId);
... lines 35 - 42
43      }
... lines 44 - 105
106 }
... lines 107 - 108
```

So this is nice: we give it the plan ID, and it gives us the corresponding, wonderful, SubscriptionPlan object:

38 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 4
class SubscriptionHelper
{
... lines 7 - 24
    /**
     * @param $planId
     * @return SubscriptionPlan|null
     */
    public function findPlan($planId)
    {
        foreach ($this->plans as $plan) {
            if ($plan->getPlanId() == $planId) {
                return $plan;
            }
        }
    }
}
```

But if the $planId doesn't exist for some reason, throw $this->createNotFoundException() to cause a 404 page:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 30
    public function addSubscriptionToCartAction($planId)
    {
        $subscriptionHelper = $this->get('subscription_helper');
        $plan = $subscriptionHelper->findPlan($planId);

        if (!$plan) {
            throw $this->createNotFoundException('Bad plan id!');
        }
... lines 39 - 42
    }
... lines 44 - 105
}

```

Finally, add the plan to the cart, with $this->get('shopping_cart')->addSubscription() and pass it the plan ID:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 30
31      public function addSubscriptionToCartAction($planId)
32      {
33          $subscriptionHelper = $this->get('subscription_helper');
34          $plan = $subscriptionHelper->findPlan($planId);
35
36          if (!$plan) {
37              throw $this->createNotFoundException('Bad plan id!');
38          }
39
40          $this->get('shopping_cart')->addSubscription($planId);
... lines 41 - 42
43      }
... lines 44 - 105
106 }
... lines 107 - 108
```

And boom! Our cart knows about the subscription! Finally, send them to the checkout page with return $this->redirectToRoute('order_checkout') - that's the name of our checkoutAction route:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 30
31      public function addSubscriptionToCartAction($planId)
32      {
33          $subscriptionHelper = $this->get('subscription_helper');
34          $plan = $subscriptionHelper->findPlan($planId);
35
36          if (!$plan) {
37              throw $this->createNotFoundException('Bad plan id!');
38          }
39
40          $this->get('shopping_cart')->addSubscription($planId);
41
42          return $this->redirectToRoute('order_checkout');
43      }
... lines 44 - 105
106 }
... lines 107 - 108
```

Adding the Subscription on the Checkout Page

Okay team, give it a try! Add the Farmer Brent plan. Bah! We need to login: use the pre-filled email and the password used by all sheep: breakingbaad.

Ok, this looks kinda right: the total is $99 because the ShoppingCart object knows about the subscription... but we haven't printed anything about the subscription in the cart table. So it looks weird.

Let's get this looking right: open the order/checkout.html.twig template and scroll down to the checkout table. We loop over the products and show the total, but never print anything about the subscription. Add a new if near the bottom: if cart - which is the ShoppingCart object - if cart.subscriptionPlan - which will be a SubscriptionPlan object or null:

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

```
... lines 1 - 59
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 64 - 66
            <div class="col-xs-12 col-sm-6">
                <table class="table table-bordered">
... lines 69 - 74
                    <tbody>
... lines 76 - 82
                        {% if cart.subscriptionPlan %}
... lines 84 - 87
                        {% endif %}
                    </tbody>
... lines 90 - 95
                </table>
            </div>
... lines 98 - 100
        </div>
    </div>
</div>
{% endblock %}
```

Then copy the <tr> from above and paste it here. Print out cart.subscriptionPlan.name:

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

```
... lines 1 - 59
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 64 - 66
            <div class="col-xs-12 col-sm-6">
                <table class="table table-bordered">
... lines 69 - 74
                    <tbody>
... lines 76 - 82
                        {% if cart.subscriptionPlan %}
                            <tr>
                                <th class="col-xs-6 checkout-product-name">Subscription: {{ cart.subscriptionPlan.name }}</th>
... line 86
                            </tr>
                        {% endif %}
                    </tbody>
... lines 90 - 95
                </table>
            </div>
... lines 98 - 100
        </div>
    </div>
</div>
{% endblock %}
```

That's why having this SubscriptionPlan object with all of those fields is really handy. Below, use cart.subscriptionPlan.price and add / month. And, whoops - I meant to use name on the first part, not price:

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

```
... lines 1 - 59
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 64 - 66
            <div class="col-xs-12 col-sm-6">
                <table class="table table-bordered">
... lines 69 - 74
                    <tbody>
... lines 76 - 82
                        {% if cart.subscriptionPlan %}
                            <tr>
                                <th class="col-xs-6 checkout-product-name">Subscription: {{ cart.subscriptionPlan.name }}</th>
                                <td class="col-xs-3">${{ cart.subscriptionPlan.price }} / month</td>
                            </tr>
                        {% endif %}
                    </tbody>
... lines 90 - 95
                </table>
            </div>
... lines 98 - 100
        </div>
    </div>
</div>
{% endblock %}
```

Let's give it a try now. It looks great! The plans are in Stripe, the plans are in our code, and you can add a plan to the cart. Time to checkout and create our first subscription.

Chapter 3: Creating the Subscription in Stripe

Open up the Stripe docs and go down the page until you find subscriptions. There's a nice little "Getting Started" section but the detailed guide is the place to go if you've got serious questions.

But let's start with the basics! Step 1... done! Step 2: subscribing your customers. Apparently all we need to do is set the plan on the Customer and save! Cool!

The Players: Subscription, Customer, Invoice

But actually, there's more going on behind the scenes. In reality, this will create a new object in Stripe: a Subscription. And actually, we're going to subscribe a user with slightly different code than this.

Keep reading below, the docs describe the *lifecycle* of a Subscription. For now, there's one *really* important thing to notice: when you create a Subscription, Stripe automatically creates an Invoice and charges that invoice immediately.

Open up the Stripe API docs so we can look at all the important objects so far. From part 1 of the tutorial, when someone buys individual products, we do a few things: we create or fetch a Customer, we create an InvoiceItem for each product and finally we create an Invoice and pay it. When you create an Invoice, Stripe automatically adds all unpaid invoice items to it.

With a Subscription, there are two new players: Plans and Subscriptions. Click "Subscriptions" and go down to "Create a Subscription". Ah, so simple: a Subscription is between a Customer and a specific Plan. This is the code we will use.

Coding up the Stripe Subscription

Back on our site, after we fill out the checkout form, the whole thing submits to OrderController::checkoutAction(). And *this* passes the submitted Stripe token to chargeCustomer():

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 48
49      public function checkoutAction(Request $request)
50      {
... lines 51 - 53
54          if ($request->isMethod('POST')) {
... lines 55 - 56
57              try {
58                  $this->chargeCustomer($token);
59              } catch (\Stripe\Error\Card $e) {
60                  $error = 'There was a problem charging your card: '.$e->getMessage();
61              }
... lines 62 - 68
69          }
... lines 70 - 77
78      }
... lines 79 - 105
106 }
... lines 107 - 108
```

Ah that's where the magic happens: it creates or gets the Customer, adds InvoiceItems and creates the Invoice:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    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);
        }

        $cart = $this->get('shopping_cart');

        foreach ($cart->getProducts() as $product) {
            $stripeClient->createInvoiceItem(
                $product->getPrice() * 100,
                $user,
                $product->getName()
            );
        }
        $stripeClient->createInvoice($user, true);
    }
}
... lines 107 - 108
```

Beautiful.

All *we* need to do is create a Subscription - via Stripe's API - if they have a plan in their cart. Before we create the Invoice, add if $cart->getSubscriptionPlan():

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    private function chargeCustomer($token)
    {
... lines 86 - 104
        if ($cart->getSubscriptionPlan()) {
... lines 106 - 113
        }
    }
}
... lines 117 - 118
```

Next, open StripeClient: we've designed this class to hold all Stripe API setup and interactions. Add a new method: createSubscription() and give it a User argument and a SubscriptionPlan argument that the User wants to subscribe to:

76 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... line 7

class StripeClient
{
... lines 11 - 65
    public function createSubscription(User $user, SubscriptionPlan $plan)
    {
... lines 68 - 73
    }
}
```

Now, go back to the Stripe API docs, steal the code that creates a Subscription, and paste it here. Set that to a new $subscription variable. For the customer, use $user->getStripeCustomerId() to get the id for *this* user. For the plan, just $plan->getPlanId(). Return the $subscription at the bottom:

76 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 65
    public function createSubscription(User $user, SubscriptionPlan $plan)
    {
        $subscription = \Stripe\Subscription::create(array(
            'customer' => $user->getStripeCustomerId(),
            'plan' => $plan->getPlanId()
        ));

        return $subscription;
    }
}
```

To use this in the controller, use the $stripeClient variable we setup earlier: $stripeClient->createSubscription() and pass it the current $user variable and then $cart->getSubscriptionPlan():

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    private function chargeCustomer($token)
    {
... lines 86 - 104
        if ($cart->getSubscriptionPlan()) {
            // a subscription creates an invoice
            $stripeClient->createSubscription(
                $user,
                $cart->getSubscriptionPlan()
            );
... lines 111 - 113
        }
    }
}
... lines 117 - 118
```

And that's *all* you need to create a subscription!

Don't Invoice Twice!

And there are no gotchas at all... oh except for this big one. Remember: when you create a Subscription, Stripe automatically creates an Invoice. And when you create an Invoice, Stripe automatically attaches all existing InvoiceItems that haven't been paid yet to that Invoice.

So, if the user has a Subscription, then an Invoice will be created when we call createSubscription(). And *that* invoice will contain any InvoiceItems for individual products that are also in the cart. If you try to create *another* invoice below, it'll be empty... and you'll actually get an error.

What we actually want to do is move createInvoice() into the else so that if there *is* a subscription plan, *it* will create the invoice, else, *we* will create it manually:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    private function chargeCustomer($token)
    {
... lines 86 - 104
        if ($cart->getSubscriptionPlan()) {
            // a subscription creates an invoice
            $stripeClient->createSubscription(
                $user,
                $cart->getSubscriptionPlan()
            );
        } else {
            // charge the invoice!
            $stripeClient->createInvoice($user, true);
        }
    }
}
... lines 117 - 118
```

Yep, the user can buy a subscription *and* some extra, amazing products all at the same time.

Try out the Whole Flow

Try the *whole* thing out: add some sheep shears to the cart so we have a product and a subscription. Fill in our fake credit card information, hit check out, and ... Cool! No errors.

But the real proof is in the dashboard. Click "Payments". Perfect! Here it is, for $124. But look closer at it, and click to view the Customer.

When we checked out, it created the customer, associated the card with it, and created an active subscription. And this was all done in this *one* invoice. It contains the subscription *plus* the one-time product purchase. In other words, this kicks butt. In one month, Stripe will automatically invoice the customer again, charge their card, and keep the subscription active.

Now that our subscription is active in Stripe, we also need to update *our* database. We need to record that this user is actively subscribed to this plan.

Chapter 4: Give the User a Subscription (in our Database)

Congrats on creating the subscription in Stripe! But now, the real work starts. Sure, Stripe knows everything about the Customer and the Subscription. But there are always going to be a few things that we need to keep in *our* database, like whether or not a user has an active subscription, and to which plan.

We're already doing this in one spot. The user table - which is modeled by this User class - has a stripeCustomerId field. Stripe holds all the customer data, but we keep track of the customer id.

We need to do the same thing for the Stripe Subscription. It also has an id, so if we can associate that with the User, we'll be able to look up that User's Subscription info.

The subscription Table

There are a few good ways to store this, but I chose to create a brand new subscription table. I'll open up a new tab in my terminal and use mysql to login to the database. fos_user is the user table and here's the new table I added: subscription.

There are a few important things. First, the subscription table has a relationship back to the user table via a user_id foreign key column. Second, the subscription table stores more than *just* the Stripe subscription id, it will also hold the planId so we can instantly know which plan a user has. It also holds a few other things that will help us manage cancellations.

So our mission is clear: when a user buys a subscription, we need to create a new row in this table, associate it with the user, and set some data on it. This will ultimately let *us* quickly determine if a user has an active subscription and to which plan.

Subscription and User Entities

The new subscription table is modeled in our code with a Subscription entity class:

95 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 4
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="subscription")
 */
class Subscription
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\OneToOne(targetEntity="User", inversedBy="subscription")
     * @ORM\JoinColumn(nullable=false, onDelete="CASCADE")
     */
    private $user;

    /**
     * @ORM\Column(type="string")
     */
    private $stripeSubscriptionId;

    /**
     * @ORM\Column(type="string")
     */
    private $stripePlanId;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $endsAt;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $billingPeriodEndsAt;
... lines 45 - 93
}
```

It has properties for all the columns you just saw. And in the User class, for convenience, I added a $subscription property shortcut:

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

```
... lines 1 - 11
class User extends BaseUser
{
... lines 14 - 35
    /**
     * @ORM\OneToOne(targetEntity="Subscription", mappedBy="user")
     */
    private $subscription;
... lines 40 - 55
    /**
     * @return Subscription
     */
    public function getSubscription()
    {
        return $this->subscription;
    }
... lines 63 - 82
}
```

With this, if you have a User object and call getSubscription() on it, you'll get the Subscription object that's associated with this User, if there is one.

Prepping the Account Page

And that's cool because we'll be able to fill in this fancy account page I created! All this info: yep, it's just hardcoded right now. Open up the template for this page at app/Resources/views/profile/account.html.twig. Instead of "None", add an if statement: if app.user - that's the currently-logged-in user app.user.subscription, then we know they have a Subscription. Add a label that says "Active". If they don't have a subscription, say "None":

49 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
... lines 6 - 11
        <div class="row">
            <div class="col-xs-6">
                <table class="table">
                    <tbody>
                        <tr>
                            <th>Subscription</th>
                            <td>
                                {% if app.user.subscription %}
                                    <span class="label label-success">Active</span>
                                {% else %}
                                    <span class="label label-default">None</span>
                                {% endif %}
                            </td>
                        </tr>
... lines 26 - 37
                    </tbody>
                </table>
            </div>
... lines 41 - 43
        </div>
    </div>
</div>
{% endblock %}
... lines 48 - 49
```

If you refresh now... it says None. We actually *do* have a Subscription in Stripe from a moment ago, but our database doesn't know about it. That's what we need to fix.

Updating the Database

Since our goal is to update the database during checkout, go back to OrderController and find the chargeCustomer() method that holds all the magic.

But instead of putting the code to update the database right here, let's add it to SubscriptionHelper: this class will do all the work related to subscriptions. Add a new method at the bottom called public function addSubscriptionToUser() with two arguments: the \Stripe\Subscription object that was just created and the User that the Subscription should belong to:

63 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 4
use AppBundle\Entity\Subscription;
use AppBundle\Entity\User;
... lines 7 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
        $subscription = $user->getSubscription();
... lines 49 - 60
    }
}
```

Inside, start with $subscription = $user->getSubscription(). So, the user may *already* have a row in the subscription table from a previous, expired subscription. If they do, we'll just update that row instead of creating a second row. Every User will have a *maximum* of one related row in the subscription table. It keeps things simple.

But if they *don't* have a previous subscription, let's create one: $subscription = new Subscription(). Then, $subscription->setUser($user):

63 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
        $subscription = $user->getSubscription();
        if (!$subscription) {
            $subscription = new Subscription();
            $subscription->setUser($user);
        }
... lines 53 - 60
    }
}
```

Our *other* todo is to update the fields on the Subscription object: $stripeSubscriptionId and $stripePlanId. To keep things clean, open Subscription and add a new method at the bottom: public function activateSubscription() with two arguments: the $stripePlanId and $stripeSubscriptionId:

102 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 94
    public function activateSubscription($stripePlanId, $stripeSubscriptionId)
    {
... lines 97 - 99
    }
}
```

Set each of these onto the corresponding properties. Also add $this->endsAt = null:

102 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 94
    public function activateSubscription($stripePlanId, $stripeSubscriptionId)
    {
        $this->stripePlanId = $stripePlanId;
        $this->stripeSubscriptionId = $stripeSubscriptionId;
        $this->endsAt = null;
    }
}
```

We'll talk more about that later, but this field will help us know whether or not a subscription has been cancelled.

Back in SubscriptionHelper, call $subscription->activateSubscription():

63 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
        $subscription = $user->getSubscription();
        if (!$subscription) {
            $subscription = new Subscription();
            $subscription->setUser($user);
        }

        $subscription->activateSubscription(
... lines 55 - 56
        );
... lines 58 - 60
    }
}
```

We need to pass this the stripePlanId and the stripeSubscriptionId. But remember! We have this fancy \Stripe\Subscription object! In the API docs, you can see its fields, like id and plan with its *own* id sub-property.

Cool! Pass the method $stripeSubscription->plan->id and $stripeSubscription->id:

63 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 53
        $subscription->activateSubscription(
            $stripeSubscription->plan->id,
            $stripeSubscription->id
        );
... lines 58 - 63
```

Booya!

And, time to save this to the database! Since we're using Doctrine in Symfony, we need the EntityManager object to do this. I'll use dependency injection: add an EntityManager argument to the __construct() method, and set it on a new $em property:

63 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 6
use Doctrine\ORM\EntityManager;

class SubscriptionHelper
{
... lines 11 - 13
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
... lines 19 - 30
    }
... lines 32 - 61
}
```

For Symfony users, this service is using auto-wiring. So because I type-hinted this with EntityManager, Symfony will automatically know to pass that as an argument.

Finally, at the bottom, add $this->em->persist($subscription) and $this->em->flush($subscription) to save *just* the Subscription:

63 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
... lines 48 - 53
        $subscription->activateSubscription(
            $stripeSubscription->plan->id,
            $stripeSubscription->id
        );

        $this->em->persist($subscription);
        $this->em->flush($subscription);
    }
}
```

With all that setup, go back to OrderController to call this method. To do that, we need the \Stripe\Subscription object. Fortunately, the createSubscription method returns this:

76 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 65
    public function createSubscription(User $user, SubscriptionPlan $plan)
    {
        $subscription = \Stripe\Subscription::create(array(
            'customer' => $user->getStripeCustomerId(),
            'plan' => $plan->getPlanId()
        ));

        return $subscription;
    }
}
```

So add $stripeSubscription = in front of that line. Then, add $this->get('subscription_helper')->addSubscriptionToUser() passing it $stripeSubscription and the currently-logged-in $user:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    private function chargeCustomer($token)
    {
... lines 86 - 104
        if ($cart->getSubscriptionPlan()) {
            // a subscription creates an invoice
            $stripeSubscription = $stripeClient->createSubscription(
                $user,
                $cart->getSubscriptionPlan()
            );

            $this->get('subscription_helper')->addSubscriptionToUser(
                $stripeSubscription,
                $user
            );
... lines 116 - 118
        }
    }
}
... lines 122 - 123
```

Phew! That may have seemed like a lot, but ultimately, this line just makes sure that there is a subscription row in our table that's associated with this user and up-to-date with the subscription and plan IDs.

Let's go try it out. Add a new subscription to your cart, fill out the fake credit card information and hit checkout. No errors! To the account page! Yes! The subscription is active! Our database is up-to-date.

Chapter 5: Data: Card Last 4 Digits

Unless I cancel my subscription... which I can't actually do yet - we'll add that soon - in 1 month, Stripe will renew my subscription by automatically charging the credit card I have on file. Eventually, we'll need to allow the user to *update* their credit card info from right here on the account page. But let's start simple: by *at least* reminding them *which* card they have on file by showing the card brand

- like VISA - and the last 4 card numbers.

This is yet *another* piece of data that's *already* stored in Stripe, but we're going to *choose* to also store it in *our* database, so we can quickly render the info to the user.

Printing the Credit Card Details

In the User class - aka our user table - I've already added two new columns: cardBrand and cardLast4:

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

```
... lines 1 - 11
class User extends BaseUser
{
... lines 14 - 25
    /**
     * @ORM\Column(type="string", nullable=true)
     */
    private $cardBrand;

    /**
     * @ORM\Column(type="string", length=4, nullable=true)
     */
    private $cardLast4;
... lines 35 - 63
    public function getCardBrand()
    {
        return $this->cardBrand;
    }

    public function setCardBrand($cardBrand)
    {
        $this->cardBrand = $cardBrand;
    }

    public function getCardLast4()
    {
        return $this->cardLast4;
    }

    public function setCardLast4($cardLast4)
    {
        $this->cardLast4 = $cardLast4;
    }
}
```

But these are empty right now: we're not actually setting this data yet.

Before we do that, let's update the template to print these fields. Open the profile/account.html.twig template. Down by the card details, let's say if app.user.cardBrand, then print some information about the user's credit card, like app.user.cardBrand ending in app.user.cardLast4:

53 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
... lines 6 - 11
        <div class="row">
            <div class="col-xs-6">
                <table class="table">
                    <tbody>
... lines 16 - 31
                        <tr>
                            <th>Credit Card</th>
                            <td>
                                {% if app.user.cardBrand %}
                                    {{ app.user.cardBrand }} ending in {{ app.user.cardLast4 }}
                                {% else %}
                                    None
                                {% endif %}
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 45 - 47
        </div>
    </div>
</div>
{% endblock %}
... lines 52 - 53
```

Those fields on the User object are empty now, so let's fix that!

The Card Details on the Stripe Customer

Head to the Stripe API docs and click on Customers. The card information is attached to the customer under a field called sources. Yes, sources with an s at the end because you *could* attach *multiple* cards to a customer if you wanted. But we're not: on checkout, we set just *one* card on the customer, and replace any existing card, if there was one.

In other words, sources will always have just one entry. That one entry will have a data key, and *that* will describe the card: giving us all the info you see here.

Now to the plan: use the Stripe API to populate the card information on the User table *right* during checkout.

Setting the Card Details

In OrderController::chargeCustomer(), we either create or retrieve the \Stripe\Customer. Assign both calls to a new $stripeCustomer variable:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 83
84      private function chargeCustomer($token)
85      {
... lines 86 - 88
89          if (!$user->getStripeCustomerId()) {
90              $stripeCustomer = $stripeClient->createCustomer($user, $token);
91          } else {
92              $stripeCustomer = $stripeClient->updateCustomerCard($user, $token);
93          }
... lines 94 - 123
124     }
125 }
... lines 126 - 127
```

In StripeClient, the createCustomer() method already returns the \Stripe\Customer object, so we're good here:

78 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
9   class StripeClient
10  {
... lines 11 - 19
20      public function createCustomer(User $user, $paymentToken)
21      {
22          $customer = \Stripe\Customer::create([
23              'email' => $user->getEmail(),
24              'source' => $paymentToken,
25          ]);
... lines 26 - 29
30
31          return $customer;
32      }
... lines 33 - 76
77  }
```

The updateCustomerCard() method, however, retrieves the customer... but gets lazy and doesn't return it. Fix that with return $customer:

78 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 33
    public function updateCustomerCard(User $user, $paymentToken)
    {
        $customer = \Stripe\Customer::retrieve($user->getStripeCustomerId());

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

        return $customer;
    }
... lines 43 - 76
}
```

Back in OrderController, we've got the \Stripe\Customer... so we're mega dangerous! But instead of updating the fields on User right here, let's do it in SubscriptionHelper. Add a new public function updateCardDetails() method with a User object that should be updated and the \Stripe\Customer object that's associated with it:

72 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 5
use AppBundle\Entity\User;
... lines 7 - 8
class SubscriptionHelper
{
... lines 11 - 62
    public function updateCardDetails(User $user, \Stripe\Customer $stripeCustomer)
    {
... lines 65 - 69
    }
}
```

Now, this is pretty easy: $cardDetails = $stripeCustomer->sources->data[0]. Then, $user->setCardBrand($cardDetails) - go cheat with the Stripe API - the fields we want are brand and last4. So, $cardDetails->brand. And $user->setCardLast4($cardDetails->last4). Save only the user to the database with the classic $this->em->persist($user) and $this->em->flush($user):

72 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 62
    public function updateCardDetails(User $user, \Stripe\Customer $stripeCustomer)
    {
        $cardDetails = $stripeCustomer->sources->data[0];
        $user->setCardBrand($cardDetails->brand);
        $user->setCardLast4($cardDetails->last4);
        $this->em->persist($user);
        $this->em->flush($user);
    }
}
```

Finally, call that method! $this->get('subscription_helper')->updateCardDetails() and pass it $user and $stripeCustomer:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    private function chargeCustomer($token)
    {
... lines 86 - 88
        if (!$user->getStripeCustomerId()) {
            $stripeCustomer = $stripeClient->createCustomer($user, $token);
        } else {
            $stripeCustomer = $stripeClient->updateCustomerCard($user, $token);
        }

        // save card details
        $this->get('subscription_helper')
            ->updateCardDetails($user, $stripeCustomer);
... lines 98 - 123
    }
}
... lines 126 - 127
```

No matter how you checkout, we're going to make sure your card details are updated in our database!

Before we try it out and *prove* how awesome we are, I want to add one more thing: I want to be able to tell the user *when* they will be billed next.

Chapter 6: So, When is my Next Invoice?

Back on the account page, our customers would *love* us if we would show them *when* they will be billed next. In other words: when will my subscription renew?

Open the Subscription class - aka the subscription table. I've already added a billingPeriodEndsAt column:

102 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 40
    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $billingPeriodEndsAt;
... lines 45 - 86
    /**
     * @return \DateTime
     */
    public function getBillingPeriodEndsAt()
    {
        return $this->billingPeriodEndsAt;
    }
... lines 94 - 100
}
```

All we need to do is set this when the subscription is first created. Ok, we also need to update it each month when the subscription is renewed - but we'll talk about that later with webhooks.

The Subscription is created - or retrieved - right here in SubscriptionHelper. This is the spot to set that date.

And hey! This method is passed a \Stripe\Subscription object:

72 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
... lines 48 - 60
    }
... lines 62 - 70
}
```

Let's check out the API docs to see what that gives us. Oh yes, it has a current_period_end property, which is a UNIX timestamp. Bingo!

Setting the billingPeriodEndsAt

In SubscriptionHelper, before we activate the subscription, add a new $periodEnd variable. To convert the timestamp to a \DateTime object, say $periodEnd = \DateTime::createFromFormat('U') - for UNIX timestamp - $stripeSubscription->current_period_end:

74 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
        $subscription = $user->getSubscription();
        if (!$subscription) {
            $subscription = new Subscription();
            $subscription->setUser($user);
        }

        $periodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);
... lines 55 - 62
    }
... lines 64 - 72
}
```

Now, pass *that* into the activateSubscription() method as a new argument:

74 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 53
        $periodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);
        $subscription->activateSubscription(
            $stripeSubscription->plan->id,
            $stripeSubscription->id,
            $periodEnd
        );
... lines 60 - 74
```

Open that function in Subscription and add a new \DateTime argument called $periodEnd. Set the property with $this->billingPeriodEndsAt = $periodEnd:

103 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 94
    public function activateSubscription($stripePlanId, $stripeSubscriptionId, \DateTime $periodEnd)
    {
... lines 97 - 98
        $this->billingPeriodEndsAt = $periodEnd;
... line 100
    }
}
```

Done!

Rendering the Next Billing Date

To celebrate, open the account.html.twig template. For "Next Billing At", add if app.user.subscription - so if they have a subscription - then print app.user.subscription.billingPeriodEndsAt|date('F jS') to format the date nicely:

57 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
... lines 6 - 11
        <div class="row">
            <div class="col-xs-6">
                <table class="table">
                    <tbody>
... lines 16 - 25
                        <tr>
                            <th>Next Billing at:</th>
                            <td>
                                {% if app.user.subscription %}
                                    {{ app.user.subscription.billingPeriodEndsAt|date('F jS') }}
                                {% else %}
                                    n/a
                                {% endif %}
                            </td>
                        </tr>
... lines 36 - 45
                    </tbody>
                </table>
            </div>
... lines 49 - 51
        </div>
    </div>
</div>
{% endblock %}
... lines 56 - 57
```

OK team! Refresh that page! The "Next Billing At" is... wrong! August 9th! That's today! But no worries, that's just because the field is blank in the database, so it's using today. To *really* test if this is working, we need to checkout with a new subscription.

Now, in real life, you probably won't allow your users to buy multiple subscriptions. Afterall, we're only storing info in the database about *one* Subscription, per user. But, for testing, it's really handy to be able to checkout over and over again.

The checkout worked! Click "Account". Yes! There is the correct date: September 9th, one month from today. And the VISA card ends in 4242.

Alright: the informational part of the account page is done. But, the user *still* needs to be able to do some pretty important stuff, like cancelling their subscription - yes, this *does* happen, it's nothing personal - and updating their credit card. Let's get to it.

Chapter 7: Canceling a Subscription

Bad news: eventually, someone will want to cancel their subscription to your amazing, awesome service. So sad. But when that happens, let's make it as *smooth* as possible. Remember: happy customers!

Like everything we do, cancelling a subscription has two parts. First we need to cancel it inside of Stripe and second, we need to update *our* database, so we know that this user no longer has a subscription.

Setting up the Cancel Button

Start by adding a cancel button to the account page. In account.html.twig, let's move the h1 down a bit:

61 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>
                    My Account
... lines 10 - 15
                </h1>
... lines 17 - 51
            </div>
... lines 53 - 55
        </div>
    </div>
</div>
{% endblock %}
... lines 60 - 61
```

Next, add a *form* with method="POST" and make this float right:

61 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>
                    My Account

                    {% if app.user.subscription %}
                        <form action="{{ path('account_subscription_cancel') }}" method="POST" class="pull-right">
                            <button type="submit" class="btn btn-danger btn-xs">Cancel Subscription</button>
                        </form>
                    {% endif %}
                </h1>
... lines 17 - 51
            </div>
... lines 53 - 55
        </div>
    </div>
</div>
{% endblock %}
... lines 60 - 61
```

We don't actually need a form, but now we can put a button inside and this will *POST* up to our server. I don't always do this right, but since this action will *change* something on the server, it's best done with a POST request. Add a few classes for styling and say "Cancel Subscription".

I still need to set the action attribute to some URL... but we need to create that endpoint first!

Open ProfileController. This file renders the account page, but we're also going to put code in here to handle some other things on this page, like cancelling a subscription and updating your credit card.

Create a new public function cancelSubscriptionAction(). Give this a URL: @Route("/profile/subscription/cancel") and a name: account_subscription_cancel:

36 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 21
    /**
     * @Route("/profile/subscription/cancel", name="account_subscription_cancel")
... line 24
     */
    public function cancelSubscriptionAction()
    {
... lines 28 - 33
    }
}
```

And, since we'll POST here, we might as well require a POST with @Method - hit tab to autocomplete and add the use statement - then POST:

36 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
... lines 6 - 11
class ProfileController extends BaseController
{
... lines 14 - 21
    /**
     * @Route("/profile/subscription/cancel", name="account_subscription_cancel")
     * @Method("POST")
     */
    public function cancelSubscriptionAction()
    {
... lines 28 - 33
    }
}
```

With the endpoint setup, copy the route name and go back into the template. Update action, with path() then paste the route:

61 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>
... lines 9 - 10
                    {% if app.user.subscription %}
                        <form action="{{ path('account_subscription_cancel') }}" method="POST" class="pull-right">
... line 13
                        </form>
                    {% endif %}
                </h1>
... lines 17 - 51
            </div>
... lines 53 - 55
        </div>
    </div>
</div>
{% endblock %}
... lines 60 - 61
```

And we are setup!

Cancel that Subscription in Stripe

Now, back to step 1: cancel the Subscription in Stripe. Go back to Stripe's documentation and find the section about Cancelling Subscriptions - it'll look a *little* different than what you see here... because Stripe updated their design *right* after I recorded. Doh! But, all the same info is there.

Ok, this is simple: retrieve a subscription and then call cancel() on it. Yes! So easy!

Cancelling at_period_end

Tip

Since 2018-07-27, Stripe changed the way you cancel a subscription at period end. Use this code for > the updated API:

```
$sub->cancel_at_period_end = true;
$sub->save();
```

Or not easy: because you *might* want to pass this an at_period_end option set to true. Here's the story: by default, when you cancel a subscription in Stripe, it cancels it immediately. But, by passing at_period_end set to true, you're saying:

> Hey! Don't cancel their subscription *now*, let them finish the month and *then* cancel it.

This is *probably* what you want: after all, your customer already paid for this month, so you'll want them to keep getting the service until its over.

So let's do this! Remember: we've organized things so that *all* Stripe API code lives inside the StripeClient object. Fetch that first with $stripeClient = $this->get('stripe_client'):

36 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
... lines 29 - 33
    }
}
```

Next, open this class, find the bottom, and add a new method: public function cancelSubscription() with one argument: the User object whose subscription should be cancelled:

89 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
... lines 80 - 86
    }
}
```

For the code inside - go copy and steal the code from the docs! Yes! Replace the hard-coded subscription id with $user->getSubscription()->getStripeSubscriptionId().

89 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
        $sub = \Stripe\Subscription::retrieve(
            $user->getSubscription()->getStripeSubscriptionId()
        );
... lines 83 - 86
    }
}
```

Then, cancel it at period end:

89 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
        $sub = \Stripe\Subscription::retrieve(
            $user->getSubscription()->getStripeSubscriptionId()
        );

        $sub->cancel([
            'at_period_end' => true,
        ]);
    }
}
```

Back in ProfileController, use this! $stripeClient->cancelSubscription() with $this->getUser() to get the currently-logged-in-user:

36 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
        $stripeClient->cancelSubscription($this->getUser());
... lines 30 - 33
    }
}
```

Then, to express how sad we are, add a heard-breaking flash message. Then, redirect back to profile_account:

36 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
        $stripeClient->cancelSubscription($this->getUser());

        $this->addFlash('success', 'Subscription Canceled :(');

        return $this->redirectToRoute('profile_account');
    }
}
```

We've done it! But don't test it yet: we still need to do step 2: update our database to reflect the cancellation.

Chapter 8: Tracking Cancelations in our Database

When the user cancels, we need to *somehow* update the user's row in the subscription table so that we know this happened! And actually, it's kind of complicated: the user canceled, but the subscription should still be active until the end of the month. *Then* it'll really be canceled. So, how the heck can we manage this?

Using Subscription endsAt

Open ProfileController. Right after we cancel the subscription in Stripe, grab the subscription object by saying, $this->getUser()->getSubscription():

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

```
... lines 1 - 11
12  class ProfileController extends BaseController
13  {
... lines 14 - 25
26      public function cancelSubscriptionAction()
27      {
... lines 28 - 30
31          $subscription = $this->getUser()->getSubscription();
... lines 32 - 39
40      }
... lines 41 - 56
57  }
```

Here's the plan: we are *not* going to delete the subscription from the subscription table, because it's still active until the period end. Instead, we'll set the endsAt date field to *when* the subscription will expire:

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
11  class Subscription
12  {
... lines 13 - 35
36      /**
37       * @ORM\Column(type="datetime", nullable=true)
38       */
39      private $endsAt;
... lines 40 - 73
74      /**
75       * @return \DateTime
76       */
77      public function getEndsAt()
78      {
79          return $this->endsAt;
80      }
81
82      public function setEndsAt(\DateTime $endsAt = null)
83      {
84          $this->endsAt = $endsAt;
85      }
... lines 86 - 123
124 }
```

That way, we'll know if the subscription is still active, meaning it's before the endsAt date, or it's fully canceled, because it's after the endsAt date.

At the bottom of Subscription, add a helper function to do this: public function deactivateSubscription():

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 102
    public function deactivateSubscription()
    {
... lines 105 - 107
    }
... lines 109 - 123
}
```

Since we know the user has paid through the end of the period, we can use that: $this->endsAt = $this->billingPeriodEndsAt. Also set $this->billingPeriodsEndsAt = null - just so we know that there won't be another bill at the end of this month:

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 102
    public function deactivateSubscription()
    {
        // paid through end of period
        $this->endsAt = $this->billingPeriodEndsAt;
        $this->billingPeriodEndsAt = null;
    }
... lines 109 - 123
}
```

Cool! To deactivate the subscription in the controller, it's as easy as saying $subscription->deactivateSubscription() and then saving it to the database with the standard persist() and flush() Doctrine code:

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

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
... lines 28 - 30
        $subscription = $this->getUser()->getSubscription();
        $subscription->deactivateSubscription();
        $em = $this->getDoctrine()->getManager();
        $em->persist($subscription);
        $em->flush();
... lines 36 - 39
    }
... lines 41 - 56
}
```

And that should do it! Let's give this guy a try. Go to the account page, then press the new "Cancel Subscription" button. Ok, looks good! Check the customer page in the Stripe dashboard. Yes! The most recent subscription - the one we're dealing

with in our code - is *active* but will cancel at the end of the month.

Showing "Canceled" on your Account

But if you look at the Account page, everything here still looks "Active". We updated the endsAt field on the subscription, but our code in this template isn't smart enough... yet.

Open account.html.twig. Hmm, I need an easy way to know whether or not a subscription is active, and if it *is* active, whether or not it's in this canceled state.

To help with this, let's create two methods inside the Subscription class. First, public function isActive():

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 109
    /**
     * Subscription is active, or cancelled but still in "active" period
     *
     * @return bool
     */
    public function isActive()
    {
... line 117
    }
... lines 119 - 123
}
```

Meaning: does the user still have an active subscription, even if it will cancel at the month's end? So, if $this->endsAt === null, then the subscription is definitely active. OR, $this->endsAt is greater than right now, new \DateTime():

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 114
    public function isActive()
    {
        return $this->endsAt === null || $this->endsAt > new \DateTime();
    }
... lines 119 - 123
}
```

Meaning the subscription is canceled, but is ending in the future.

The second method we need is public function isCanceled():

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 119
    public function isCancelled()
    {
... line 122
    }
}
```

Meaning: if the subscription is active, has the user actually canceled it or not? This will simply be, return $this->endsAt !== null:

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 119
    public function isCancelled()
    {
        return $this->endsAt !== null;
    }
}
```

Oh man, our setup is getting fancy! Let's get even fancier with one more helper method, this time in User. Add a new public function hasActiveSubscription():

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

```
... lines 1 - 11
class User extends BaseUser
{
... lines 14 - 83
    public function hasActiveSubscription()
    {
        return $this->getSubscription() && $this->getSubscription()->isActive();
    }
... lines 88 - 92
}
```

A User has an active subscription if they have a subscription object related to them and that subscription object isActive(). That'll save us some typing whenever we need to check whether or not a user has an active subscription.

Making the Account Template Awesome

Ok, back to the account template! This time, to be heros!

First, that "Cancel Subscription" button should only be there if the user has an active subscription. No problem! Add if app.user.hasActiveSubscription():

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>
... lines 9 - 10
                    {% if app.user.hasActiveSubscription %}
... lines 12 - 16
                        <form action="{{ path('account_subscription_cancel') }}" method="POST" class="pull-right">
                            <button type="submit" class="btn btn-danger btn-xs">Cancel Subscription</button>
                        </form>
... line 20
                    {% endif %}
                </h1>
... lines 23 - 63
            </div>
... lines 65 - 67
        </div>
    </div>
</div>
... lines 71 - 73
```

But even here, if the user has already *canceled* their subscription, we don't want to keep showing them this button. Add another if: if app.user.subscription.isCancelled():

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>
... lines 9 - 10
                    {% if app.user.hasActiveSubscription %}
                        {% if app.user.subscription.isCancelled %}
... lines 13 - 15
                        {% else %}
                            <form action="{{ path('account_subscription_cancel') }}" method="POST" class="pull-right">
                                <button type="submit" class="btn btn-danger btn-xs">Cancel Subscription</button>
                            </form>
                        {% endif %}
                    {% endif %}
                </h1>
... lines 23 - 63
            </div>
... lines 65 - 67
        </div>
    </div>
</div>
{% endblock %}
... lines 72 - 73
```

Then add a little "TODO" to add a re-activate button. If they've cancelled, they might remember how cool your service is and want to come back LATER and reactivate! In the else, show them the Cancel button.

Finish up the endif and the other endif. And actually, copy these first two lines: we need to re-use them further below. In the section that tells us whether or not we have an active subscription, we now have three states: "active", "active but canceled," and "none." Replace the old if statement with the two that you just copied. If the subscription is canceled, add label-warning and say "Canceled". Else, we know it's active:

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 8 - 23
                <table class="table">
                    <tbody>
                        <tr>
                            <th>Subscription</th>
                            <td>
                                {% if app.user.hasActiveSubscription %}
                                    {% if app.user.subscription.isCancelled %}
                                        <span class="label label-warning">Cancelled</span>
... lines 32 - 33
                                    {% else %}
                                        <span class="label label-success">Active</span>
                                    {% endif %}
                                {% else %}
                                    <span class="label label-default">None</span>
                                {% endif %}
                            </td>
                        </tr>
... lines 42 - 61
                    </tbody>
                </table>
            </div>
... lines 65 - 67
        </div>
    </div>
</div>
{% endblock %}
... lines 72 - 73
```

If a user doesn't have any type of active subscription, keep the "none" from before.

Finally, copy *just* the first if statement and scroll down to "Next Billing at". We should *only* show the next billing period if the user has an *active* subscription, not just if they have a related subscription object, because it could be canceled. Paste the if statement over this one:

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 8 - 23
                <table class="table">
                    <tbody>
... lines 26 - 41
                        <tr>
                            <th>Next Billing at:</th>
                            <td>
                                {% if app.user.hasActiveNonCancelledSubscription %}
                                    {{ app.user.subscription.billingPeriodEndsAt|date('F jS') }}
                                {% else %}
                                    n/a
                                {% endif %}
                            </td>
                        </tr>
... lines 52 - 61
                    </tbody>
                </table>
            </div>
... lines 65 - 67
        </div>
    </div>
</div>
{% endblock %}
... lines 72 - 73
```

Finally, do the same thing down below for the credit card: I don't want to confuse someone by showing them credit card information when they don't have a subscription:

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 8 - 23
                <table class="table">
                    <tbody>
... lines 26 - 51
                        <tr>
                            <th>Credit Card</th>
                            <td>
                                {% if app.user.hasActiveNonCancelledSubscription %}
                                    {{ app.user.cardBrand }} ending in {{ app.user.cardLast4 }}
                                {% else %}
                                    None
                                {% endif %}
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 65 - 67
        </div>
    </div>
</div>
{% endblock %}
... lines 72 - 73
```

Phew! Ok, refresh! It's beautiful! There's our todo for the reactivate button and the subscription is canceled. But wait! We don't want the "Next Billing at" and credit card information to show up.

Ah, that's my bad! The hasActiveSubscription() returns true *even* if the user already cancelled it. Open User: let's add one more method: public function hasActiveNonCanceledSubscription():

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

```
... lines 1 - 11
class User extends BaseUser
{
... lines 14 - 88
    public function hasActiveNonCancelledSubscription()
    {
... line 91
    }
}
```

Inside, return $this->hasActiveSubscription() && !$this->getSubscription()->isCancelled():

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

```
... lines 1 - 11
12  class User extends BaseUser
13  {
... lines 14 - 88
89      public function hasActiveNonCancelledSubscription()
90      {
91          return $this->hasActiveSubscription() && !$this->getSubscription()->isCancelled();
92      }
93  }
```

Use this method in both places in the Twig template:

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
3   {% block body %}
4   <div class="nav-space">
5       <div class="container">
6           <div class="row">
7               <div class="col-xs-6">
... lines 8 - 23
24                  <table class="table">
25                      <tbody>
... lines 26 - 41
42                          <tr>
43                              <th>Next Billing at:</th>
44                              <td>
45                                  {% if app.user.hasActiveNonCancelledSubscription %}
46                                      {{ app.user.subscription.billingPeriodEndsAt|date('F jS') }}
47                                  {% else %}
48                                      n/a
49                                  {% endif %}
50                              </td>
51                          </tr>
52                          <tr>
53                              <th>Credit Card</th>
54                              <td>
55                                  {% if app.user.hasActiveNonCancelledSubscription %}
56                                      {{ app.user.cardBrand }} ending in {{ app.user.cardLast4 }}
57                                  {% else %}
58                                      None
59                                  {% endif %}
60                              </td>
61                          </tr>
62                      </tbody>
63                  </table>
64              </div>
... lines 65 - 67
68          </div>
69      </div>
70  </div>
71  {% endblock %}
... lines 72 - 73
```

Refresh one more time! We got it!

But now that the user can cancel, let's make it possible for them to *reactivate* the subscription. It's actually an easy win.

Chapter 9: Reactivate/Un-cancel my Subscription!

So, if someone cancels, they can't *un-cancel*. And that's a bummer!

In the Stripe API docs, under the "canceling" section, there's actually a spot about reactivating canceled subscriptions, and it's really interesting! It says that if you use the at_period_end method of canceling, and the subscription has *not* yet reached the period end, then reactivating is easy: just set the subscription's plan to the same plan ID that it had originally. Internally, Stripe knows that means I want to *not* cancel the subscription anymore.

Route and Controller Setup

Let's hook it up! We're going to need a new endpoint that reactivates a subscription. In ProfileController, add a new public function reactivateSubscriptionAction(). Give it a route set to /profile/subscription/reactivate and a name: account_subscription_reactivate:

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

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 41
    /**
     * @Route("/profile/subscription/reactivate", name="account_subscription_reactivate")
     */
    public function reactivateSubscriptionAction()
    {
... lines 47 - 55
    }
}
```

Good start! With this in place, copy the route name, open account.html.twig and go up to the "TODO" we added a few minutes ago. Paste the route, just to stash it somewhere, then copy the entire *cancel* form and put it here. Update the form action with the new route name, change the text, and use btn-success to make this look like a really happy thing:

73 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>
                    My Account

                    {% if app.user.hasActiveSubscription %}
                        {% if app.user.subscription.isCancelled %}
                            <form action="{{ path('account_subscription_reactivate') }}" method="POST" class="pull-right">
                                <button type="submit" class="btn btn-success btn-xs">Reactivate Subscription</button>
                            </form>
                        {% else %}
                            <form action="{{ path('account_subscription_cancel') }}" method="POST" class="pull-right">
                                <button type="submit" class="btn btn-danger btn-xs">Cancel Subscription</button>
                            </form>
                        {% endif %}
                    {% endif %}
                </h1>
... lines 23 - 63
            </div>
... lines 65 - 67
        </div>
    </div>
</div>
{% endblock %}
... lines 72 - 73
```

Refresh and enjoy the nice, new Reactivate Subscription button. Beautiful!

Expired Subscriptions Cannot be Reactivated

Let's get to work in the controller. Like everything, this will have two parts. First, we need to reactivate the subscription in Stripe and second, we need to update our database. For the first part, fetch the trusty StripeClient service object with $stripeClient = $this->get('stripe_client'):

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

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 44
    public function reactivateSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
... lines 48 - 55
    }
}
```

Next, open that class. Add a new public function reactivateSubscription(). It will need a User argument whose subscription we should reactivate:

105 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 88
    public function reactivateSubscription(User $user)
    {
... lines 91 - 102
    }
}
```

As the Stripe docs mentioned, we can only reactivate a subscription that has *not* been fully canceled. If today is *beyond* the period end, then the user will need to create an entirely new subscription. That's why we only show the button in our template during this period.

But just in case, add an "if" statement: if !$user->hasActiveSubscription(), then we'll throw a new exception with the text:

> Subscriptions can only be reactivated if the subscription has not actually ended.

105 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 88
    public function reactivateSubscription(User $user)
    {
        if (!$user->hasActiveSubscription()) {
            throw new \LogicException('Subscriptions can only be reactivated if the subscription has not actually ended yet');
        }
... lines 94 - 102
    }
}
```

Nothing should hit that code, but now we'll know if something does.

Reactivate in Stripe

To reactivate the Subscription, we first need to fetch it. In the Stripe API docs, find "Retrieve a Subscription." Every object can be fetched using the same retrieve method. Copy this. Then, add, $subscription = and paste. Replace the subscription ID with $user->getSubscription()->getStripeSubscriptionId():

105 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 90
        if (!$user->hasActiveSubscription()) {
            throw new \LogicException('Subscriptions can only be reactivated if the subscription has not actually ended yet');
        }

        $subscription = \Stripe\Subscription::retrieve(
            $user->getSubscription()->getStripeSubscriptionId()
        );
... lines 98 - 105
```

And remember, if any API call to Stripe fails - like because this is an invalid subscription ID - the library will throw an exception. So we don't need to add extra code to check if that subscription was found.

Finally, reactivate the subscription by setting its plan property equal to the original plan ID, which is $user->getSubscription()->getStripePlanId():

105 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 90
        if (!$user->hasActiveSubscription()) {
            throw new \LogicException('Subscriptions can only be reactivated if the subscription has not actually ended yet');
        }

        $subscription = \Stripe\Subscription::retrieve(
            $user->getSubscription()->getStripeSubscriptionId()
        );
        // this triggers the refresh of the subscription!
        $subscription->plan = $user->getSubscription()->getStripePlanId();
... lines 100 - 105
```

Then, send the details to Stripe with $subscription->save():

105 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 90
        if (!$user->hasActiveSubscription()) {
            throw new \LogicException('Subscriptions can only be reactivated if the subscription has not actually ended yet');
        }

        $subscription = \Stripe\Subscription::retrieve(
            $user->getSubscription()->getStripeSubscriptionId()
        );
        // this triggers the refresh of the subscription!
        $subscription->plan = $user->getSubscription()->getStripePlanId();
        $subscription->save();
... lines 101 - 105
```

And just in case, return the $subscription:

105 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 88
    public function reactivateSubscription(User $user)
    {
        if (!$user->hasActiveSubscription()) {
            throw new \LogicException('Subscriptions can only be reactivated if the subscription has not actually ended yet');
        }

        $subscription = \Stripe\Subscription::retrieve(
            $user->getSubscription()->getStripeSubscriptionId()
        );
        // this triggers the refresh of the subscription!
        $subscription->plan = $user->getSubscription()->getStripePlanId();
        $subscription->save();

        return $subscription;
    }
}
```

Love it! Back in ProfileController, reactivate the subscription with, $stripeClient->reactivateSubscription($this->getUser()):

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

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 44
    public function reactivateSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
        $stripeSubscription = $stripeClient->reactivateSubscription($this->getUser());
... lines 49 - 55
    }
}
```

And we are done on the Stripe side.

Updating our Database

The other thing we need to worry about - which turns out to be really easy - is to update our database so that this, once again, looks like an active subscription. It's easy, because we've already done the work for this. Check out SubscriptionHelper: we have a method called addSubscriptionToUser(), which is normally used right after the user originally buys a new subscription:

74 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
        $subscription = $user->getSubscription();
        if (!$subscription) {
            $subscription = new Subscription();
            $subscription->setUser($user);
        }

        $periodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);
        $subscription->activateSubscription(
            $stripeSubscription->plan->id,
            $stripeSubscription->id,
            $periodEnd
        );

        $this->em->persist($subscription);
        $this->em->flush($subscription);
    }
... lines 64 - 72
}
```

But we can could also call this after reactivating. In reality, this method simply ensures that the Subscription row in the table is up-to-date with the latest stripePlanId, stripeSubscriptionId, periodEnd and endsAt:

74 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
... lines 48 - 54
        $subscription->activateSubscription(
            $stripeSubscription->plan->id,
            $stripeSubscription->id,
            $periodEnd
        );
... lines 60 - 62
    }
... lines 64 - 72
}
```

These last two are the most important: because they changed when we deactivated the subscription. So by calling activateSubscription():

125 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 94
    public function activateSubscription($stripePlanId, $stripeSubscriptionId, \DateTime $periodEnd)
    {
        $this->stripePlanId = $stripePlanId;
        $this->stripeSubscriptionId = $stripeSubscriptionId;
        $this->billingPeriodEndsAt = $periodEnd;
        $this->endsAt = null;
    }
... lines 102 - 123
}
```

All of that will be reversed, and the subscription will be alive!

Let's do it! In ProfileController, add a $stripeSubscription = in front of the $stripeClient call. Below that, use $this->get('subscription_helper')->addSubscriptionToUser() and pass it $stripeSubscription and the current user:

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

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 44
    public function reactivateSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
        $stripeSubscription = $stripeClient->reactivateSubscription($this->getUser());

        $this->get('subscription_helper')
            ->addSubscriptionToUser($stripeSubscription, $this->getUser());
... lines 52 - 55
    }
}
```

And that is everything!

Give your user a happy flash message and redirect back to the profile page:

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

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 44
    public function reactivateSubscriptionAction()
    {
        $stripeClient = $this->get('stripe_client');
        $stripeSubscription = $stripeClient->reactivateSubscription($this->getUser());

        $this->get('subscription_helper')
            ->addSubscriptionToUser($stripeSubscription, $this->getUser());

        $this->addFlash('success', 'Welcome back!');

        return $this->redirectToRoute('profile_account');
    }
}
```

I think we're ready to try this! Go back and refresh the profile. Press reactivate and... our "cancel subscription" button is back, "active" is back, "next billing period" is back and "credit card" is back. In Stripe, the customer's most recent subscription also became active again. Oh man, this is kind of fun to play with: cancel, reactivate, cancel, reactivate. The system is solid.

Chapter 10: Cancelation Edge-Case Bugs

I hope you now think that canceling and reactivating feels pretty easy! Well, it is! Except for 2 minor, edge-case bugs that have caused us problems in the past. Let's fix them now.

Problem 1: Canceling Past Due Accounts

First, go to the Stripe API docs and go down to subscription. You'll notice that one of the fields is called status, which has a number of different values. The most important ones for us are active, past_due, which means it's still in an active state, but we're having problems charging their card, and canceled.

Here's problem number 1: at the end of the month, Stripe will try to charge your user for the renewal. To do that, it will create an invoice and then charge that invoice. If, for some reason, the user's credit card can't be charged, the invoice remains created and Stripe will try to charge that invoice a few more times. That's something we'll talk a lot more about in a few minutes.

Now, imagine that the invoice has been created and we're having problems charging the user's credit card. Then, the user goes to our site and cancels. Since we're canceling "at period end", the invoice in Stripe won't be deleted, and Stripe will continue to try to charge that invoice a few more times. In other words, we will attempt to charge a user's credit card, after they cancel! Not cool!

To fix this, we need to *fully* cancel the user's subscription. That will close the invoice and stop future payment attempts on it.

Squashing the Bug: Fully Cancel

In StripeClient::cancelSubscription(), it's time to squash this bug. First, create a new variable called $cancelAtPeriodEnd and set it to true. Then, down below, set the at_period_end option to this variable:

118 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
... lines 80 - 84
        $cancelAtPeriodEnd = true;
... lines 86 - 94
        $sub->cancel([
            'at_period_end' => $cancelAtPeriodEnd,
        ]);
... lines 98 - 99
    }
... lines 101 - 116
}
```

Now, here's the trick: if $subscription->status == 'past_due', then it means that the invoice *has* been created and we're having problems charging it. In this case, set $cancelAtPeriodEnd to false:

118 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
... lines 80 - 84
        $cancelAtPeriodEnd = true;

        if ($sub->status == 'past_due') {
            // past due? Cancel immediately, don't try charging again
            $cancelAtPeriodEnd = false;
... lines 90 - 92
        }

        $sub->cancel([
            'at_period_end' => $cancelAtPeriodEnd,
        ]);
... lines 98 - 99
    }
... lines 101 - 116
}
```

This will cause the subscription to cancel immediately and close that invoice!

Problem 2: Canceling within 1 Hour of Renewal

But there's one other, weirder, but similar problem. At the end of the month, 1 hour before charging the user, Stripe creates the invoice. It then waits 1 hour, and tries to charge the user for the first time. So, if your user cancels *within* that hour, then we also need to fully cancel that subscription to prevent its invoice from being paid.

This is a little trickier: we basically need to see if the user is canceling within that one hour window. To figure that out, create a new variable called $currentPeriodEnd and set that to a \new DateTime() with the @ symbol and $subscription->current_period_end:

118 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
... lines 80 - 83
        $currentPeriodEnd = new \DateTime('@'.$sub->current_period_end);
        $cancelAtPeriodEnd = true;
... lines 86 - 99
    }
... lines 101 - 116
}
```

This converts that timestamp into a \DateTime object.

Now, if $currentPeriodEnd < new \DateTime('+1 hour'), then this means that we're probably in that window and should set $cancelAtPeriodEnd = false:

118 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
... lines 80 - 83
        $currentPeriodEnd = new \DateTime('@'.$sub->current_period_end);
        $cancelAtPeriodEnd = true;

        if ($sub->status == 'past_due') {
            // past due? Cancel immediately, don't try charging again
            $cancelAtPeriodEnd = false;
        } elseif ($currentPeriodEnd < new \DateTime('+1 hour')) {
            // within 1 hour of the end? Cancel so the invoice isn't charged
            $cancelAtPeriodEnd = false;
        }

        $sub->cancel([
            'at_period_end' => $cancelAtPeriodEnd,
        ]);
... lines 98 - 99
    }
... lines 101 - 116
}
```

An easy way of thinking of this is, if the user is *pretty* close to the end of their period, then canceling now versus at period end, is almost the same. So, we'll just be careful.

But for this to work, your server's timezone needs to be set to UTC, which is the timezone used by the timestamps sent back from Stripe. If you're not sure, you could give yourself some more breathing room, but fully-canceling anyone's subscription that is within one day of the period end.

Fully Canceling in the Database

These fixes created a *new* problem! Now, when the user clicks the "Cancel Subscription" button, we *might* be canceling the subscription *right* now, and we need to update the database to reflect that.

To do that, first return the $stripeSubscription from the cancelSubscription() method:

118 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 77
    public function cancelSubscription(User $user)
    {
... lines 80 - 94
        $sub->cancel([
            'at_period_end' => $cancelAtPeriodEnd,
        ]);

        return $sub;
    }
... lines 101 - 116
}
```

Then, in ProfileController, add $stripeSubscription = before the cancelSubscription() call:

65 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
... line 28
        $stripeSubscription = $stripeClient->cancelSubscription($this->getUser());

        $subscription = $this->getUser()->getSubscription();
... lines 32 - 46
    }
... lines 48 - 63
}
```

Finally, we can use the status field to know whether or not the subscription has truly been canceled, or if it's still active until the period end. In other words, if $stripeSubscription->status == 'canceled', then the subscription is done! Else, we're canceling at period end and should just call deactivate():

65 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
... line 28
        $stripeSubscription = $stripeClient->cancelSubscription($this->getUser());

        $subscription = $this->getUser()->getSubscription();

        if ($stripeSubscription->status == 'canceled') {
            // the subscription was cancelled immediately
... line 35
        } else {
            $subscription->deactivateSubscription();
        }
... lines 39 - 46
    }
... lines 48 - 63
}
```

To handle full cancelation, open up Subscription and add a new public function called cancel(). Here, set $this->endsAt to right now, to guarantee that it will look canceled, and $this->billingPeriodEndsAt = null:

131 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 109
    public function cancel()
    {
        $this->endsAt = new \DateTime();
        $this->billingPeriodEndsAt = null;
    }
... lines 115 - 129
}
```

In ProfileController, call it: $subscription->cancel():

65 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 25
    public function cancelSubscriptionAction()
    {
... lines 28 - 32
        if ($stripeSubscription->status == 'canceled') {
            // the subscription was cancelled immediately
            $subscription->cancel();
        } else {
            $subscription->deactivateSubscription();
        }
... lines 39 - 46
    }
... lines 48 - 63
}
```

And we are done!

Now, testing this is a bit difficult. So let's just make sure we didn't break anything major by hitting cancel. Perfect! And we can reactivate.

And *this* is why subscriptions are hard.

Chapter 11: The Update Card Form!

Eventually, our customers will need to *update* the credit card that we have stored. And on surface, this is pretty easy: the card is just a property on the Stripe customer called source. And basically, we need to *nearly* duplicate the checkout process: show the user a card form, exchange their card info for a Stripe token, submit that token, and then save it on the user as their *new* card.

Ok, let's rock! Step 1: add an "Update Card" button to the account page and make it re-use the same card form we already built for checkout. Because, ya know, reusing code is awesome!

Re-using the Card Form

In account.html.twig, find the credit card section. Add a button, give it some classes for styling *and* a js-open-credit-card-form class:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 16
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 22 - 37
                <table class="table">
                    <tbody>
... lines 40 - 65
                        <tr>
                            <th>Credit Card</th>
                            <td>
                                {% if app.user.hasActiveNonCancelledSubscription %}
                                    {{ app.user.cardBrand }} ending in {{ app.user.cardLast4 }}

                                    <button class="btn btn-xs btn-link pull-right js-open-credit-card-form">
                                        Update Card
                                    </button>
                                {% else %}
                                    None
                                {% endif %}
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 83 - 88
        </div>
    </div>
</div>
{% endblock %}
... lines 93 - 94
```

In a second, we'll attach some JavaScript to this.

Next, find the col-xs-6:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 16
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
... lines 21 - 82
            <div class="col-xs-6">
... lines 84 - 87
            </div>
        </div>
    </div>
</div>
{% endblock %}
... lines 93 - 94
```

This is the *right* side of the page, and I've kept it empty until now *just* so we can show the form here. Add a div with a JS class so we can hide/show this element: js-update-card-wrapper. Make it display: none; by default:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 16
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
... lines 21 - 82
            <div class="col-xs-6">
                <div class="js-update-card-wrapper" style="display: none;">
... lines 85 - 86
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
... lines 93 - 94
```

Inside, add a cute header:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 16
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
... lines 21 - 82
            <div class="col-xs-6">
                <div class="js-update-card-wrapper" style="display: none;">
                    <h2>Update Credit Card</h2>
... line 86
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
... lines 93 - 94
```

Ok, I want to re-use the entire checkout form right here. Fortunately, we already isolated that into its own template: _cardForm.html.twig. Yay!

In account.html.twig, use the Twig include() function to bring that in:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 16
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
... lines 21 - 82
            <div class="col-xs-6">
                <div class="js-update-card-wrapper" style="display: none;">
                    <h2>Update Credit Card</h2>
                    {{ include('order/_cardForm.html.twig') }}
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
... lines 93 - 94
```

Hide/Show the Form

Ok! Let's hide/show this form whenever the user clicks the "Update Card" button. At the top of the file, override the block called javascripts, and call endblock. Inside, call the parent() function:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}
... lines 5 - 14
{% endblock %}
... lines 16 - 94
```

In this project, any JS we put here will be included on the page.

Add a script tag and a very simple document.ready() block:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
... lines 8 - 12
        });
    </script>
{% endblock %}
... lines 16 - 94
```

Inside of that, find the .js-open-credit-card-form element and on click, create a callback function. Start with the normal e.preventDefault():

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
            $('.js-open-credit-card-form').on('click', function(e) {
                e.preventDefault();
... lines 10 - 11
            });
        });
    </script>
{% endblock %}
... lines 16 - 94
```

Now, find the other wrapper element, which is js-update-card-wrapper. Call slideToggle() on that to show/hide it:

94 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
            $('.js-open-credit-card-form').on('click', function(e) {
                e.preventDefault();

                $('.js-update-card-wrapper').slideToggle();
            });
        });
    </script>
{% endblock %}
... lines 16 - 94
```

So, fairly easy stuff.

Well, maybe we should see if it works first. Refresh! Ah, it doesn't! Huge error:

> Variable "error" does not exist in _cardForm.html.twig at line 56

Hmm, checkout that template:

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>
```

Ah yes, on the checkout page, after we submit, if there was an error, we set this variable and render it here. For now, we don't have any errors. In account.html.twig, we could pass an error variable to the include() call. But, we could also do it in ProfileController::accountAction(). Add error set to null:

67 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 16
    public function accountAction()
    {
        return $this->render('profile/account.html.twig', [
            'error' => null
        ]);
    }
... lines 23 - 65
}
```

Refresh and click "Update Card". We are in business!

Fixing the Button Text

But, this has two cosmetic problems. First, the button says "Checkout"! That's a little scary, and misleading. Let's change it!

In _cardForm.html.twig, replace "Checkout" with a new variable called buttonText|default('Checkout'):

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">
                {{ buttonText|default('Checkout') }}
            </button>
        </div>
    </div>
</form>
```

So, if the variable is *not* defined, it'll print "Checkout".

Now, override that in account.html.twig. Give include() a second argument: an array of extra variables. Pass buttonText as

Update Card:

96 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 16
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
... lines 21 - 82
            <div class="col-xs-6">
                <div class="js-update-card-wrapper" style="display: none;">
                    <h2>Update Credit Card</h2>
                    {{ include('order/_cardForm.html.twig', {
                        buttonText: 'Update Card'
                    }) }}
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
... lines 95 - 96
```

Refresh! Cool, button problem solved.

Sharing Card JavaScript

The second problem is pretty obvious if you fill out the card number field on checkout, and then compare it with the profile page! On the checkout page, we included a lot of JavaScript that does cool stuff like format this field. And, *much* more importantly, the JS is also responsible for sending the credit card information to Stripe, fetching the token, putting it in the form, and submitting it. We *definitely* still need that.

Ok, how can we reuse the JavaScript? In checkout.html.twig, we just inlined all of our JS right in the template. That's not great, but since this isn't a course about JavaScript, let's solve this as easily as possible. Copy all of the JS and create a new template called _creditCardFormJavaScript.html.twig inside the order/ directory. Paste this there:

51 lines | app/Resources/views/order/_creditCardFormJavaScript.html.twig

```
<script type="text/javascript" src="https://js.stripe.com/v2/"></script>
<script src="{{ asset('js/jquery.payment.min.js') }}"></script>

<script type="text/javascript">
    Stripe.setPublishableKey('{{ stripe_public_key }}');

    $(function () {
        var $form = $('.js-checkout-form');

        $form.find('.js-cc-number').payment('formatCardNumber');
        $form.find('.js-cc-exp').payment('formatCardExpiry');
        $form.find('.js-cc-cvc').payment('formatCardCVC');

        $form.submit(function (event) {
            event.preventDefault();

            // Disable the submit button to prevent repeated clicks:
            $form.find('.js-submit-button').prop('disabled', true);

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

    function stripeResponseHandler(status, response) {
        // Grab the form:
        var $form = $('.js-checkout-form');

        if (response.error) { // Problem!

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

        } else { // Token was created!
            $form.find('.js-checkout-error')
                .addClass('hidden');

            // 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>
```

Now, in checkout, include that template!

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

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

    {{ include('order/_creditCardFormJavaScript.html.twig') }}
{% endblock %}
... lines 9 - 55
```

Copy that and include the same thing in account.html.twig at the top of the javascripts block:

98 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    {{ include('order/_creditCardFormJavaScript.html.twig') }}
... lines 7 - 16
{% endblock %}
... lines 18 - 98
```

Ok, refresh and hope for the best! Ah, another missing variable: stripe_public_key:

> Variable "stripe_public_key" does not exist in _creditCardFormJavaScript.html.twig at line 5

We're printing this in the middle of our JS:

51 lines | app/Resources/views/order/_creditCardFormJavaScript.html.twig

```
... lines 1 - 3
<script type="text/javascript">
    Stripe.setPublishableKey('{{ stripe_public_key }}');
... lines 6 - 50
</script>
```

And the variable comes from OrderController. Copy that line, open ProfileController, and paste it there:

68 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 11
class ProfileController extends BaseController
{
... lines 14 - 16
    public function accountAction()
    {
        return $this->render('profile/account.html.twig', [
... line 20
            'stripe_public_key' => $this->getParameter('stripe_public_key'),
        ]);
    }
... lines 24 - 66
}
```

Now the page works! *And* - at the very least - the JS formatting is rocking.

Frontend stuff is done: let's submit this and update the user's card in Stripe.

Chapter 12: Saving the Updated Card Details

When the user fills out this form, our JavaScript sends that info to Stripe, Stripe then sends back a token, we add the token as a hidden field in the form and then submit it. Both the checkout form and update card form will work like this. But, we need to submit the update card form to a new endpoint, that'll *update* the card, but *not* charge the user.

Submitting the Form to the new Endpoint

Open ProfileController and add a new endpoint: public function updateCreditCardAction(). Give it the URL /profile/card/update and a fancy name: account_update_credit_card. Add the @Method("POST") to be extra cool:

93 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
... lines 9 - 12
class ProfileController extends BaseController
{
... lines 15 - 68
    /**
     * @Route("/profile/card/update", name="account_update_credit_card")
     * @Method("POST")
     */
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 90
    }
}
```

With this in place, we need to update the form to submit here. Check out _cardForm.html.twig. Hmm, the action attribute is *empty*:

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

```
<form action="" method="POST" class="js-checkout-form checkout-form">
... lines 2 - 66
</form>
```

That's because we want the checkout form to submit right back to /checkout. But this won't work for the update card form: it should submit to a different URL.

Instead, render a new variable called formAction and pipe that to the default filter with empty quotes:

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

```
<form action="{{ formAction|default('') }}" method="POST" class="js-checkout-form checkout-form">
... lines 2 - 66
</form>
```

Now we can override this! In account.html.twig, add another variable to the include: formAction set to path() and the new route name:

99 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 18
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
... lines 23 - 84
            <div class="col-xs-6">
                <div class="js-update-card-wrapper" style="display: none;">
                    <h2>Update Credit Card</h2>
                    {{ include('order/_cardForm.html.twig', {
                        buttonText: 'Update Card',
                        formAction: path('account_update_credit_card')
                    }) }}
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
... lines 98 - 99
```

Refresh and check out the source. Ok, the form action is ready!

Saving the Credit Card

Let's get to work in ProfileController! But actually... this will be very similar to our checkout logic, so let's go steal code! Copy the line that fetches the stripeToken POST parameter and then head back to ProfileController. Make sure you have a Request argument and the Request use statement:

93 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
... lines 9 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 90
    }
}
```

Then, first, paste that line to fetch the token. Second, fetch the current user object with $user = $this->getUser(). And third, we need to make an API request to stripe that updates the Customer and attaches the token as their new card. That means we'll be using the StripeClient. Fetch it first with $stripeClient = $this->get('stripe_client'):

93 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
        $token = $request->request->get('stripeToken');
        $user = $this->getUser();

        $stripeClient = $this->get('stripe_client');
... lines 79 - 90
    }
}
```

Here's the awesome part: open StripeClient. We *already* have a method called updateCustomerCard():

118 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 33
    public function updateCustomerCard(User $user, $paymentToken)
    {
        $customer = \Stripe\Customer::retrieve($user->getStripeCustomerId());

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

        return $customer;
    }
... lines 43 - 116
}
```

We pass the User object and the submitted payment token and *it* sets it on the Customer and saves.

Victory for code organization! In the controller, just say $stripeClient->updateCustomerCard() and pass it $user and $token:

93 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 77
        $stripeClient = $this->get('stripe_client');
        $stripeCustomer = $stripeClient->updateCustomerCard(
            $user,
            $token
        );
... lines 83 - 90
    }
}
```

That takes care of things inside Stripe.

Updating cardLast4 and cardBrand

Now, what about our database? Do we store any information about the credit card? Actually, we do! In the User class, we store cardLast4 and cardBrand. With the new card, this stuff probably changed!

But wait, we've got this handled too guys! Open SubscriptionHelper and check out the handy updateCardDetails() method:

74 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 64
    public function updateCardDetails(User $user, \Stripe\Customer $stripeCustomer)
    {
        $cardDetails = $stripeCustomer->sources->data[0];
        $user->setCardBrand($cardDetails->brand);
        $user->setCardLast4($cardDetails->last4);
        $this->em->persist($user);
        $this->em->flush($user);
    }
}
```

Just pass it the User and \Stripe\Customer and it'll make sure those fields are set.

In ProfileController, call this: $this->get('subscription_helper')->updateCardDetails() passing $user and $stripeCustomer... which doesn't exist yet. Fortunately, updateCustomerCard() returns that, so create that variable on that line:

93 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 78
        $stripeCustomer = $stripeClient->updateCustomerCard(
            $user,
            $token
        );

        // save card details!
        $this->get('subscription_helper')
            ->updateCardDetails($user, $stripeCustomer);
... lines 87 - 90
    }
}
```

That's it! Add a success message so that everyone feels happy and joyful, and redirect back to the profile page:

93 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 83
        // save card details!
        $this->get('subscription_helper')
            ->updateCardDetails($user, $stripeCustomer);

        $this->addFlash('success', 'Card updated!');

        return $this->redirectToRoute('profile_account');
    }
}
```

Time to try it! Refresh and put in the fake credit card info. But use a different expiration: 10/25. Hit "Update Card". Ok, it looks like it worked. Refresh the Customer page in Stripe. The expiration *was* 10/20 and now it's 10/25. Card update successful!

But, we still need to handle one more case: when the card update fails.

Chapter 13: Handling Card Update Fails

There are actually two ways for the credit card update to fail. Most failures happen immediately, and are handle via JavaScript. But a few don't happen until we try to attach the card to the customer.

Let's see an example. In Stripe's documentation, find the "Testing" section - it's under the Payments header in the new design. Down the page a bit, you'll find a table full of cards that will work or fail for different reasons. Find the one that ends in 0002 and copy it. Fill the form out using this and update the card.

Ah, 500 error!

> Your card was declined

Ok, a \Stripe\Error\Card exception was thrown the moment that we tried to save the new customer card back to Stripe. We *did* handle this situation on our checkout page, so we just need to *also* handle it here.

In ProfileController, the updateCustomerCard() call is the one that might fail. Wrap this is a try-catch for \Stripe\Error\Card:

101 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 77
        try {
            $stripeClient = $this->get('stripe_client');
            $stripeCustomer = $stripeClient->updateCustomerCard(
                $user,
                $token
            );
        } catch (\Stripe\Error\Card $e) {
... lines 85 - 89
        }
... lines 91 - 99
}
```

Set an $error variable to: There was a problem charging your card and then concatenate $e->getMessage():

101 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 77
        try {
            $stripeClient = $this->get('stripe_client');
            $stripeCustomer = $stripeClient->updateCustomerCard(
                $user,
                $token
            );
        } catch (\Stripe\Error\Card $e) {
            $error = 'There was a problem charging your card: '.$e->getMessage();
... lines 86 - 89
        }
... lines 91 - 99
}
```

To show this to the user, call addFlash() and set an error type:

101 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 77
        try {
            $stripeClient = $this->get('stripe_client');
            $stripeCustomer = $stripeClient->updateCustomerCard(
                $user,
                $token
            );
        } catch (\Stripe\Error\Card $e) {
            $error = 'There was a problem charging your card: '.$e->getMessage();

            $this->addFlash('error', $error);
... lines 88 - 89
        }
... lines 91 - 99
}
```

Just like with success flash messages, our base template is already configured to show these. But in this case, the message will look red and angry!

Finally, redirect back to the profile_account route:

101 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 72
    public function updateCreditCardAction(Request $request)
    {
... lines 75 - 77
        try {
            $stripeClient = $this->get('stripe_client');
            $stripeCustomer = $stripeClient->updateCustomerCard(
                $user,
                $token
            );
        } catch (\Stripe\Error\Card $e) {
            $error = 'There was a problem charging your card: '.$e->getMessage();

            $this->addFlash('error', $error);

            return $this->redirectToRoute('profile_account');
        }
... lines 91 - 99
}
```

To try it out, go back, press "Update Card" again, and use the same, failing card number. This time, no 500 error! Just this sad, but useful message.

With the ability to subscribe, cancel and update their credit card info, our subscription system is up-and-running! Now it's time to face our *last* big, *required* topic: webhooks. How can we email a user if we're having problems charging their card? How would we know when Stripe cancel's a customer's subscription due to payment failure? And how can we email a receipt each month when a subscription renews?

The answer to all of these is: webhooks. And getting those right will make your system *really* rock.

Chapter 14: Stripe Events & Webhooks

Oh boy, it's *finally* time to talk about the big subject I've been avoiding: webhooks. You see, every month Stripe is going to *try* to renew each customer's subscription. If it fails to charge their card, you might want to send them an email. If it's successful, you might want to send them a receipt. Or, if it fails so many times that the subscription needs to be cancelled, then we need to update our database to reflect that.

In short, Stripe needs to communicate back to *us* when certain things, or *events* happen.

Stripe Events

Go to our Customer page in Stripe. At the bottom, you'll see an *events* section. Basically, whenever *anything* happens in Stripe, an event is created. For example, when we added a new card, there was an event whose type field is set to customer.source.created. Every action becomes an *event* and there are many different *types* of events for the many things that can happen.

In fact, switch over to Stripe's API docs. Event is so important that it's an *object* in the API: you can list and fetch them. Click Types of events. Awesome! A big, giant, beautiful list of all the different event types so you can figure out what each event means.

What Happens when we Can't Charge a User?

Out of this list, there are just a few types that you'll need to worry about. The first is the event type that occurs when the customer's subscription is canceled when Stripe can't charge their card for renewal.

So, what *actually* happens when Stripe can't charge a card? Go back to the Stripe Dashboard and go to "Account Settings", and then "Subscriptions". This screen is *really* important: it determines *exactly* what happens when a card can't be charged. By default, Stripe will attempt to charge the card once, then try again 3, 5 and 7 days later. If it *still* can't charge the card, it will finally cancel the subscription. You can tweak the timing, but the story is always this: Stripe tries to charge a few times, then eventually cancels the subscription.

Hello Webhooks & requestb.in

When this happens, we need Stripe to tell us that the subscription was canceled. And we'll do this via a webhook. It's pretty simple: we configure a webhook URL to our site in Stripe. Then, whenever certain event *types* happen, Stripe will send a request to our URL that contains the *event* object as JSON. So if Stripe sent us a webhook whenever a subscription was canceled, we would be in business!

A really nice way to test out webhooks is by using a site called http://requestb.in.

Tip

The http://requestb.in site is no longer available (see https://github.com/Runscope/requestbin#readme). Try https://requestbin.com instead that has a bunch more features.

Click "Create a RequestBin". Ok, real simple: this page will record and display any requests made to this temporary URL.

Back on our dashboard, add a webhook and paste the URL. Put it in test mode, so that we only receive events from the "test" environment. Next, click "Select events". Instead of receiving *all* event types, let's just choose the few we want. For now, that's just customer.subscription.deleted.

Yep, this is the event that happens when a subscription is cancelled, for any reason, including when a user's card couldn't be charged.

Create that webhook! Ok, let's see what a test webhook looks like. Click "Send test webhook" and change the event to customer.subscription.deleted. Now, send that webhook!

Refresh the RequestBin page. So cool! This shows us the raw JSON request body that was just sent to us. These events are just objects in Stripe's API, like Customer, Subscription or anything else. But if you configure a webhook, then Stripe will

send that event to *us*, instead of us needing to fetch it from them.

Here's the next goal: setup a route and controller on our site that's capable of handling webhooks and doing different things in our system when different event types happen.

Chapter 15: Webhook Endpoint Setup

Let's get right to work on our webhook endpoint. In the src/AppBundle/Controller directory, create a new WebhookController class. Make it extend a BaseController class I created - that just has a few small shortcuts:

18 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 2
namespace AppBundle\Controller;
... lines 4 - 8
class WebhookController extends BaseController
{
... lines 11 - 17
}
```

Now, create the endpoint with public function stripeWebhookAction(). Give an @Route annotation set to /webhooks/stripe and a similar name. Make sure you have the Route use statement:

18 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Request;
... lines 7 - 8
class WebhookController extends BaseController
{
    /**
     * @Route("/webhooks/stripe", name="webhook_stripe")
     */
    public function stripeWebhookAction(Request $request)
    {
... line 16
    }
}
```

Start simple: return a new Response() from the HttpFoundation component:

18 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 6
use Symfony\Component\HttpFoundation\Response;
... line 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
        return new Response('baaaaaa');
    }
}
```

That's just enough to try it out: find your browser and go to /webhooks/stripe. It's alive!

Decoding the Event

Thanks to RequestBin, we know more or less what the JSON body will look like. The most important thing is this *event* id.

Let's decode the JSON and grab this.

To do that, add $data = json_decode(), but pause there. We need to pass this the *body* of the Request. In Symfony, we get this by adding a Request argument - don't forget the use statement! Then, use $request->getContent(). Also, pass true as the second argument so that json_decode returns an associative array:

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 5
use Symfony\Component\HttpFoundation\Request;
... lines 7 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);
... lines 17 - 34
    }
}
```

Next, it *shouldn't* happen, but just in case, if $data is null, that means Stripe sent us invalid JSON. Shame on you Stripe! Throw an exception in this case... and make sure you spell Exception correctly!

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 5
use Symfony\Component\HttpFoundation\Request;
... lines 7 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);
        if ($data === null) {
            throw new \Exception('Bad JSON body from Stripe!');
        }
... lines 20 - 34
    }
}
```

Finally, get the $eventId from $data['id']:

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 5
use Symfony\Component\HttpFoundation\Request;
... lines 7 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);
        if ($data === null) {
            throw new \Exception('Bad JSON body from Stripe!');
        }

        $eventId = $data['id'];
... lines 22 - 34
    }
}
```

We Found the Event! Now, Fetch the Event?!

Ok, let's refocus on the next steps. Ultimately, I want to read these fields in the event, find the Subscription in the database, and cancel it. But instead of reading the JSON body directly, we're going to use Stripe's API to fetch the Event object by using this $eventId.

Wait, but won't that just return the *exact* same data we already have? Yes! We do this not because we *need* to, but for security. If we read the request JSON directly, it's possible that the request is coming from some external, mean-spirited person instead of from Stripe. By fetching a fresh event from Stripe, it guarantees the event data is legitimate.

Since we make all API requests through the StripeClient class, open it up and scroll to the bottom. Add a new public function called findEvent() with an $eventId argument. Inside, just return \Stripe\Event::retrieve() and pass it $eventId:

127 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 117
    /**
     * @param $eventId
     * @return \Stripe\Event
     */
    public function findEvent($eventId)
    {
        return \Stripe\Event::retrieve($eventId);
    }
}
```

Back in the controller, add $stripeEvent = $this->get('stripe_client')->findEvent($eventId):

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 20
        $eventId = $data['id'];

        $stripeEvent = $this->get('stripe_client')
            ->findEvent($eventId);
... lines 25 - 34
    }
}
```

If this were an invalid event ID, Stripe would throw an exception.

With that, we're prepped to handle some event types.

Chapter 16: Webhook: Subscription Canceled

Eventually, this function will handle *several* event types. To handle each, create a switch-case statement: switch ($stripeEvent->type):

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 22
        $stripeEvent = $this->get('stripe_client')
            ->findEvent($eventId);

        switch ($stripeEvent->type) {
... lines 27 - 31
        }
... lines 33 - 34
    }
}
```

That's the field that'll hold one of those *many* event types we saw earlier.

The *first* type we'll handle is customer.subscription.deleted. We'll fill in the logic here in a second. Add the break:

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 22
        $stripeEvent = $this->get('stripe_client')
            ->findEvent($eventId);

        switch ($stripeEvent->type) {
            case 'customer.subscription.deleted':
                // todo - fully cancel the user's subscription
                break;
... lines 30 - 31
        }
... lines 33 - 34
    }
}
```

We shouldn't receive *any* other event types because of how we configured the webhook, but just in case, throw an Exception: "Unexpected webhook from Stripe" and pass the type:

36 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 22
        $stripeEvent = $this->get('stripe_client')
            ->findEvent($eventId);

        switch ($stripeEvent->type) {
            case 'customer.subscription.deleted':
                // todo - fully cancel the user's subscription
                break;
            default:
                throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 33 - 34
    }
}
```

At the bottom, well, we can return *anything* back to Stripe. How about a nice message: "Event Handled" and then the type. Well, there is *one* important piece: you *must* return a 200-level status code. If you return a *non* 200 status code, Stripe will think the webhook failed and will try to send it again, over and over again. But 200 means:

> Yo Stripe, it's cool - I heard you, I handled it.

Quick! Cancel the Subscription!

Alright, let's cancel the subscription! First, we need to find the Subscription in our database. And check this out: the subscription id lives at data.object.id. That's because *this* type of event embeds the subscription in question. *Other* event types will embed *different* data.

Add $stripeSubscriptionId = $stripeEvent->data->object->id:

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 26
        switch ($stripeEvent->type) {
            case 'customer.subscription.deleted':
                $stripeSubscriptionId = $stripeEvent->data->object->id;
... lines 30 - 33
                break;
            default:
                throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 38 - 39
    }
... lines 41 - 60
}
```

Next, the subscription table has a stripeSubscriptionId field on it. Let's query on this! Because I already know I'll want to re-use this next code, I'll put the logic into a private function. On this line, call that future function with $subscription = $this->findSubscription() and pass it $stripeSubscriptionId:

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 26
        switch ($stripeEvent->type) {
            case 'customer.subscription.deleted':
                $stripeSubscriptionId = $stripeEvent->data->object->id;
                $subscription = $this->findSubscription($stripeSubscriptionId);
... lines 31 - 33
                break;
            default:
                throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 38 - 39
    }
... lines 41 - 60
}
```

Scroll down and create this: private function findSubscription() with its $stripeSubscriptionId argument:

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 41
    /**
     * @param $stripeSubscriptionId
     * @return \AppBundle\Entity\Subscription
     * @throws \Exception
     */
    private function findSubscription($stripeSubscriptionId)
    {
... lines 49 - 59
    }
}
```

Query by adding $subscription = $this->getDoctrine()->getRepository('AppBundle:Subscription') and then findOneBy() passing this an array with one item: stripeSubscriptionId - the field name to query on - set to $stripeSubscriptionId:

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 46
    private function findSubscription($stripeSubscriptionId)
    {
        $subscription = $this->getDoctrine()
            ->getRepository('AppBundle:Subscription')
            ->findOneBy([
                'stripeSubscriptionId' => $stripeSubscriptionId
            ]);

        if (!$subscription) {
            throw new \Exception('Somehow we have no subscription id ' . $stripeSubscriptionId);
        }
... lines 58 - 59
    }
}
```

If there is *no* matching Subscription... well, that shouldn't happen! But just in case, throw a new Exception with a really confused message. Something is not right.

Finally, return the $subscription on the bottom:

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 46
    private function findSubscription($stripeSubscriptionId)
    {
        $subscription = $this->getDoctrine()
            ->getRepository('AppBundle:Subscription')
            ->findOneBy([
                'stripeSubscriptionId' => $stripeSubscriptionId
            ]);

        if (!$subscription) {
            throw new \Exception('Somehow we have no subscription id ' . $stripeSubscriptionId);
        }

        return $subscription;
    }
}
```

Ok, head back up to the action method. Hmm, so all we really need to do now is call the cancel() method on $subscription! But let's get a *little* bit more organized. Open SubscriptionHelper and add a new method there:
public function fullyCancelSubscription() with the Subscription object that should be canceled. Below, really simple, say $subscription->cancel(). Then, use the Doctrine entity manager to save this to the database:

81 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 73
    public function fullyCancelSubscription(Subscription $subscription)
    {
        $subscription->cancel();
        $this->em->persist($subscription);
        $this->em->flush($subscription);
    }
}
```

Mind blown!

Back in the controller, call this! Above the switch statement, add a $subscriptionHelper variable set to $this->get('subscription_helper'):

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 25
        $subscriptionHelper = $this->get('subscription_helper');
        switch ($stripeEvent->type) {
... lines 28 - 36
        }
... lines 38 - 39
    }
... lines 41 - 60
}
```

Finally, call $subscriptionHelper->fullyCancelSubscription($subscription):

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 25
        $subscriptionHelper = $this->get('subscription_helper');
        switch ($stripeEvent->type) {
            case 'customer.subscription.deleted':
                $stripeSubscriptionId = $stripeEvent->data->object->id;
                $subscription = $this->findSubscription($stripeSubscriptionId);

                $subscriptionHelper->fullyCancelSubscription($subscription);

                break;
            default:
                throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }

        return new Response('Event Handled: '.$stripeEvent->type);
    }
... lines 41 - 60
}
```

And that is it!

Yep, there's some setup to get the webhook controller started, but now we're in really good shape.

Of course... we have no way to test this... So, ya know, just make sure you do a *really* good job of coding and hope for the best! No, that's crazy!, I'll show you a few ways to test this next. But also, don't forget to configure your webhook URL in Stripe once you finally deploy this to beta and production. I have a webhook setup for each instance on KnpU.

Chapter 17: Testing Webhooks

The reason that webhooks are so hard is... well, they're kind of impossible to test. There are certain things on Stripe - like my card being declined during a renewal, which are *really* hard to simulate. And even if we could, Stripe can't send a webhook to my local computer. Well actually, that last part isn't entirely true, but more on that later.

So let's look at a few strategies for making sure that your webhooks are *air tight*. I do *not* want to mess something up with these.

Automated Test

The first strategy - and the one we use here on KnpU - is to create an automated test that sends a fake webhook to the URL.

To start, let's install PHPUnit into the project:

```
$ composer require phpunit/phpunit --dev
```

While Jordi is working on that, go back to the code and find a tutorials directory. This is a special directory I created and you should have it if you downloaded the start code from this page. It has a few things to make our life easier.

Copy the WebhookControllerTest.php file and put it into the tests/AppBundle/Controller directory. Let's check this out:

45 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 2
namespace Tests\AppBundle\Controller;

use AppBundle\Entity\Subscription;
use AppBundle\Entity\User;
use Doctrine\ORM\EntityManager;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class WebhookControllerTest extends WebTestCase
{
    private $container;
    /** @var EntityManager */
    private $em;

    public function setUp()
    {
        self::bootKernel();
        $this->container = self::$kernel->getContainer();
        $this->em = $this->container->get('doctrine')->getManager();
    }

    private function createSubscription()
    {
        $user = new User();
        $user->setEmail('fluffy'.mt_rand().'@sheep.com');
        $user->setUsername('fluffy'.mt_rand());
        $user->setPlainPassword('baa');

        $subscription = new Subscription();
        $subscription->setUser($user);
        $subscription->activateSubscription(
            'plan_STRIPE_TEST_ABC'.mt_rand(),
            'sub_STRIPE_TEST_XYZ'.mt_rand(),
            new \DateTime('+1 month')
        );

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

        return $subscription;
    }
}
```

This is the *start* of a test that's specifically written for Symfony. If you're not using Symfony, the code will look different, but the idea is fundamentally the same.

Testing Strategy

This test boots Symfony's kernel so that we have access to its container, and all the useful objects inside, like the entity manager! I also added a private function called createSubscription(). We're not using this yet, but by calling it, it will create a new user in the database, give that user an active subscription, and save everything. This won't be a *real* subscription in Stripe - it'll just live in our database, and will have a random stripeSubscriptionId.

Because here's the strategy:

1. Create a fake User and fake Subscription in the database;
2. Send a webhook to /webhooks/stripe with *fake* JSON, where the subscription id in the JSON matches the fake data in the database;
3. Verify that the subscription is fully canceled after the webhook finishes.

Add the test: public function testStripeCustomerSubscriptionDeleted():

54 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 22
    public function testStripeCustomerSubscriptionDeleted()
    {
... lines 25 - 29
    }
... lines 31 - 52
}
```

OK, step 1: create the subscription in the database: $subscription = $this->createSubscription(). Easy! Step 2: send the webhook... which I'll put as a TODO for now. Then step 3: $this->assertFalse() that $subscription->isActive():

54 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 22
    public function testStripeCustomerSubscriptionDeleted()
    {
        $subscription = $this->createSubscription();

        // todo - send the cancellation webhook

        $this->assertFalse($subscription->isActive());
    }
... lines 31 - 52
}
```

Make sense?

Prepping some Fake JSON

To send the webhook, we first need to prepare a JSON string that matches what Stripe sends. At the bottom of the class, create a new private function called getCustomerSubscriptionDeletedEvent() with a $subscriptionId argument:

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 67
    private function getCustomerSubscriptionDeletedEvent($subscriptionId)
    {
... lines 70 - 122
    }
}
```

To fill this in, go copy the *real* JSON from the test webhook. Paste it here with $json = <<<EOF enter, and paste!

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 67
    private function getCustomerSubscriptionDeletedEvent($subscriptionId)
    {
        $json = <<<EOF
{
  "created": 1326853478,
  "livemode": false,
  "id": "evt_00000000000000",
  "type": "customer.subscription.deleted",
  "object": "event",
  "request": null,
  "pending_webhooks": 1,
  "api_version": "2016-07-06",
  "data": {
    "object": {
      "id": "%s",
      "object": "subscription",
      "application_fee_percent": null,
      "cancel_at_period_end": true,
      "canceled_at": 1469731697,
      "created": 1469729305,
      "current_period_end": 1472407705,
      "current_period_start": 1469729305,
      "customer": "cus_00000000000000",
      "discount": null,
      "ended_at": 1470436151,
      "livemode": false,
      "metadata": {
      },
      "plan": {
        "id": "farmer_00000000000000",
        "object": "plan",
        "amount": 9900,
        "created": 1469720306,
        "currency": "usd",
        "interval": "month",
        "interval_count": 1,
        "livemode": false,
        "metadata": {
        },
        "name": "Farmer Brent (monthly)",
        "statement_descriptor": null,
        "trial_period_days": null
      },
      "quantity": 1,
      "start": 1469729305,
      "status": "canceled",
      "tax_percent": null,
      "trial_end": null,
      "trial_start": null
```

```
117         }
118     }
119 }
120 EOF;
    ... lines 121 - 122
123         }
124 }
```

Now here's the important part: our controller reads the data.object.id key to find the subscription id. Replace this with %s:

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
    ... lines 1 - 9
10  class WebhookControllerTest extends WebTestCase
11  {
    ... lines 12 - 67
68      private function getCustomerSubscriptionDeletedEvent($subscriptionId)
69      {
70          $json = <<<EOF
71  {
    ... lines 72 - 79
80      "data": {
81        "object": {
82          "id": "%s",
    ... lines 83 - 94
95          },
    ... lines 96 - 116
117     }
118   }
119 }
120 EOF;
    ... lines 121 - 122
123     }
124 }
```

Then, finish the function with return sprintf($json, $subscriptionId):

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
    ... lines 1 - 9
10  class WebhookControllerTest extends WebTestCase
11  {
    ... lines 12 - 67
68      private function getCustomerSubscriptionDeletedEvent($subscriptionId)
69      {
70          $json = <<<EOF
    ... lines 71 - 119
120 EOF;
121
122         return sprintf($json, $subscriptionId);
123     }
124 }
```

Now, this function will create a realistic-looking JSON string, but with whatever subscriptionId we want!

Back in the test function, add $eventJson = $this->getCustomerSubscriptionDeletedEvent() and pass it $subscription->getStripeSubscriptionId(), which is some fake, random value that the function below created:

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 22
    public function testStripeCustomerSubscriptionDeleted()
    {
        $subscription = $this->createSubscription();

        $eventJson = $this->getCustomerSubscriptionDeletedEvent(
            $subscription->getStripeSubscriptionId()
        );
... lines 30 - 42
        $this->assertFalse($subscription->isActive());
    }
... lines 45 - 123
}
```

Sending the Fake Webhook

To send the request, create a $client variable set to $this->createClient(). This is Symfony's internal HTTP client: its job is to make requests to our app. If you want, you can also use something different, like Guzzle. It doesn't really matter because - one way or another - you just need to make an HTTP request to the endpoint.

Now for the magic: call $client->request() and pass it a bunch of arguments: POST for the HTTP method, /webhooks/stripe, then a few empty arrays for parameters, files and server. Finally, for the $content argument - the *body* of the request - pass it $eventJson:

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 22
    public function testStripeCustomerSubscriptionDeleted()
    {
        $subscription = $this->createSubscription();

        $eventJson = $this->getCustomerSubscriptionDeletedEvent(
            $subscription->getStripeSubscriptionId()
        );

        $client = $this->createClient();
        $client->request(
            'POST',
            '/webhooks/stripe',
            [],
            [],
            [],
            $eventJson
        );
... lines 40 - 42
        $this->assertFalse($subscription->isActive());
    }
... lines 45 - 123
}
```

And because things almost never work for me on the first try... and because I *know* this won't work yet, let's dump($client->getResponse()->getContent()) to see what happened in case there's an error. Also add a sanity check, $this->assertEquals() that 200 matches $client->getResponse()->getStatusCode():

125 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 22
    public function testStripeCustomerSubscriptionDeleted()
    {
... lines 25 - 31
        $client->request(
            'POST',
            '/webhooks/stripe',
            [],
            [],
            [],
            $eventJson
        );
        dump($client->getResponse()->getContent());
        $this->assertEquals(200, $client->getResponse()->getStatusCode());

        $this->assertFalse($subscription->isActive());
    }
... lines 45 - 123
}
```

Let's run the test! But not in this video... this video is getting too long. So go get some more coffee and then come back. Then, to the test!

Chapter 18: Testing Part 2: Faking the Event Lookup

Let's run the test. Copy its method name, then open your terminal. It looks like PHPUnit installed just fine. So, run:

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

Oh no! It blew up! Hmmm:

> Unknown database 'stripe_recording_test'

Ah, my bad!

I setup our project to use a *different* database for testing... and I forgot to create it! Do that with:

```
$ ./bin/console doctrine:database:create --env=test
```

And to create the tables, run:

```
$ ./bin/console doctrine:schema:create --env=test
```

Try the test again:

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

Another error! Scroll to the top! The webhook returned a *500* error. And if you look closely at the dumped response HTML, you can see the reason:

> No such event: evt_00000000000000

Ah, the id of the fake event that we're sending is evt_00000000000000. That's *not* a real event in Stripe, and so when the WebhookController reads this and uses Stripe's API to *fetch* this event, it's not there:

61 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 22
        $stripeEvent = $this->get('stripe_client')
            ->findEvent($eventId);
... lines 25 - 39
    }
... lines 41 - 60
}
```

It's kind of funny: we added this API lookup to prevent a third-party from sending fake events... and now it's stopping us from doing *exactly* that. Dang!

Faking things in the Test Environment

Hmm, how to fix this? In the real world, we *do* want to use stripe's API to fetch the Event object. But in the test environment, this would all work if our code would simply use the JSON we're sending it as the event, and skip the lookup.

Let's do it! We'll set a special configuration variable in the *test* environment only, then use that to change our logic in the controller.

Open app/config/config.yml and add a new parameter: verify_stripe_event set to true:

```
80 lines | app/config/config.yml
... lines 1 - 7
parameters:
... line 9
    verify_stripe_event: true
... lines 11 - 80
```

Copy that, and open config_test.yml. Add a parameters key, paste this parameter, but override it to be false:

```
25 lines | app/config/config_test.yml
... lines 1 - 3
parameters:
    verify_stripe_event: false
... lines 6 - 25
```

Now, in WebhookController, we just need an if statement: if $this->getParameter('verify_stripe_event') is true, then keep the normal behavior. Otherwise, set $stripeEvent to json_decode($request->getContent()):

```
66 lines | src/AppBundle/Controller/WebhookController.php
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 22
        if ($this->getParameter('verify_stripe_event')) {
            $stripeEvent = $this->get('stripe_client')
                ->findEvent($eventId);
        } else {
            // fake the Stripe_Event in the test environment
            $stripeEvent = json_decode($request->getContent());
        }
... lines 30 - 44
    }
... lines 46 - 65
}
```

OK, this is not *technically* perfect: the first $stripeEvent is a \Stripe\Event object, and the second will be an instance of stdClass. But, since you fetch data off both the same way, it should work.

Let's see if does! Try the test again:

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

This time, no errors! And the dumped response content looks perfect: event handled.

Refreshing the Data after the Test

But, the test didn't pass:

> Failed asserting that true is false on line 43

It looks like our webhook is *not* working, because the subscription is still active. But actually, that's not true: Doctrine is tricking us! In reality, the database *has* been updated to show that the Subscription is canceled, but this Subscription object is out-of-date. Query for a fresh one with $subscription = $this->em - I set the EntityManager on that property in setup() - then ->getRepository('AppBundle:Subscription') with find($subscription->getId()):

128 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 22
    public function testStripeCustomerSubscriptionDeleted()
    {
... lines 25 - 31
        $client->request(
            'POST',
            '/webhooks/stripe',
            [],
            [],
            [],
            $eventJson
        );
        $this->assertEquals(200, $client->getResponse()->getStatusCode());

        $subscription = $this->em
            ->getRepository('AppBundle:Subscription')
            ->find($subscription->getId());

        $this->assertFalse($subscription->isActive());
    }
... lines 48 - 126
}
```

This subscription will have *fresh* data.

Try the test!

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

And we are green!

I know, that was kind of hard! But if you want to have automated webhook tests, this is the way to do it. To make matters worse, for other webhooks, you may need to fake *additional* API calls that you're making to Stripe.

But, there are also a couple of other, manual, but easy ways to test. Let's check 'em out!

Chapter 19: Live Webhook Testing with Ngrok

If the automated tests aren't your thing, or if you just *really* want to see your webhooks in action... in real life, you've got two options.

Testing on Beta

The best way, honestly, is to test your webhooks for real, out in the wild, on a beta server. Yep, I simply mean: deploy your code to beta, setup a webhook to point to your beta server, and then start creating test subscriptions through your site.

The only problem is that some webhooks are harder to simulate than others. Want to simulate customer.subscription.deleted? No problem: create a subscription on your site, then log into Stripe and cancel it. Then watch the webhook magic happen.

But faking the invoice.payment_failed webhook because your card is being declined on subscription renewal... well... that's a bit harder. You *could* wait 1 month and see what happens... if you have a lot of time. Or, you could temporarily create a *new* subscription plan and set its interval to one day.

Then, you can test a different situation each day: like make sure the subscription renewal webhook works, then update your card to one that will fail, wait one more day, and see how your system handles the invoice.payment_failed webhook.

It's not perfect, it's slow, but it's totally real-world.

Using your Local Machine with Ngrok

The second option is to point a webhook at your *local* development machine. But wait! That's not possible: our local machine is not accessible by the internet.

Well... that doesn't *have* to be true. By using a cool utility called Ngrok, you can temporarily tunnel a *public* URL to your computer.

Let's try it! Since I already have ngrok installed, I can use it from any directory on my system:

```
$ ngrok http 8000
```

That will expose port 8000 - the one we're serving our site on - to the web via this cool public URL.

Copy that and paste it into your browser! Ah, that's a little security check that prevents any non-local users from accessing our dev environment. Just for now, go into the web/ directory, open app_dev.php, and comment-out the two security lines:

33 lines | web/app_dev.php

```
... lines 1 - 12
if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !(in_array(@$_SERVER['REMOTE_ADDR'], ['127.0.0.1', 'fe80::1', '::1']) || php_sapi_name() === 'cli-server')
) {
//    header('HTTP/1.0 403 Forbidden');
//    exit('You are not allowed to access this file. Check '.basename(__FILE__).' for more information.');
}
... lines 20 - 33
```

Refresh again! Hey, it's our site! Via a public URL.

Using the Ngrok URL as a Webhook Endpoint

Now we're super dangerous! In your Stripe Webhook configuration, add an endpoint. Paste the URL and put this in the "Test" environment. For now, *just* receive the customer.subscription.deleted event. Create the endpoint!

Oh, wait, make sure the endpoint URL ends in /webhooks/stripe. That's better!

In our app, we *already* have an active subscription. So, in Stripe, click "Customers" and open our one Customer. Find the top subscription and... cancel it! Immediately!

That *should* cause a webhook to be sent to our local machine. So, moment of truth: refresh the account page! Oh no! The subscription doesn't look canceled!

Hmm, go back to Stripe. At the bottom of the Customer page, you can see all the events for this Customer. Or, another way to look at this is by clicking "Events & webhooks" on the left. Ah! And we *can* see the customer.subscription.deleted event! And at the bottom, it shows that the webhook to our ngrok URL *was* successful.

So, refresh the account page again. Ah, *now* the subscription is canceled. We were just *too* fast the first time.

So, choose your favorite method of testing these crazy webhook things, and make sure they are bug-free.

Chapter 20: Webhook: Email User on Subscription Renewal

The second webhook *type* we need to handle is called invoice.payment_succeeded. This one fires when a subscription is successfully *renewed*. Well, actually, it fires whenever *any* invoice is paid, but we'll sort that out later.

This webhook is important to us for 2 reasons.

First, each subscription has a $billingPeriodEndsAt value that we use to show the user when Stripe will charge them next:

131 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
11  class Subscription
12  {
... lines 13 - 40
41      /**
42       * @ORM\Column(type="datetime", nullable=true)
43       */
44      private $billingPeriodEndsAt;
... lines 45 - 129
130 }
```

Obviously, that needs to be updated each month!

Second, when you charge your customer, you should probably send them a nice email about it, and maybe even attach a receipt! So let's get this setup.

Receive all Webhook Types

Right now, Stripe is *not* sending us this webhook type. In the dashboard, update the RequestBin webhook and set it to receive *all* webhooks, instead of just the few that we select. You don't *need* to do this, but it does make it easier to keep your various webhooks - like for your staging and production servers - identical.

But now we need to do some work in WebhookController. In the default section of the switch-case, we *will* now receive unsupported webhooks, and that's cool! Remove the exception:

79 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
9   class WebhookController extends BaseController
10  {
... lines 11 - 13
14      public function stripeWebhookAction(Request $request)
15      {
... lines 16 - 31
32          switch ($stripeEvent->type) {
... lines 33 - 50
51              default:
52                  // allow this - we'll have Stripe send us everything
53                  // throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
54          }
... lines 55 - 56
57      }
... lines 58 - 77
78  }
```

Inspecting the invoice.payment_succeeded Webhook

Each webhook type has a JSON body that looks a little different. Head back to the dashboard to send a test webhook, this time for the invoice.payment_succeeded event . Hit "Send test webhook" and then go refresh RequestBin.

Hmm, ok. This time, the embedded object is an invoice. But *we* will need to know the Stripe subscription ID that this invoice is for. And that's tricky: an invoice may *not* actually contain a subscription. If you just buy some products on our site, that creates an invoice... but with no subscription.

Fortunately, the data key covers this: it has a subscription field. This will either be blank if there's no subscription or it will hold the subscription ID. In other words, it's perfect!

Handling invoice.payment_succeeded

Back in WebhookController and add a second case statement invoice.payment_succeeded. Add the break, then let's get to work:

79 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 39
            case 'invoice.payment_succeeded':
... lines 41 - 49
                break;
            default:
                // allow this - we'll have Stripe send us everything
                // throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 55 - 56
    }
... lines 58 - 77
}
```

First, grab the subscription ID with $stripeSubscriptionId = $stripeEvent->data->object->subscription:

79 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 39
            case 'invoice.payment_succeeded':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;
... lines 42 - 49
                break;
            default:
                // allow this - we'll have Stripe send us everything
                // throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 55 - 56
    }
... lines 58 - 77
}
```

Next, if there *is* a $stripeSubscriptionId, then we need to load the corresponding Subscription from our database. Re-use $this->findSubscription() from earlier to do that:

79 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 39
            case 'invoice.payment_succeeded':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;

                if ($stripeSubscriptionId) {
                    $subscription = $this->findSubscription($stripeSubscriptionId);
... lines 45 - 48
                }
                break;
            default:
                // allow this - we'll have Stripe send us everything
                // throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 55 - 56
    }
... lines 58 - 77
}
```

Remember: the goal is to update this Subscription row to have the *new* billingPeriodEndsAt. But the event's data doesn't have that date! No problem: if we fetch a fresh, full Subscription object from Stripe's API, we can use its current_period_end field.

Open up StripeClient and add a new public function findSubscription() with a $stripeSubscriptionId argument:

136 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 126
    /**
     * @param $stripeSubscriptionId
... line 129
     */
    public function findSubscription($stripeSubscriptionId)
    {
... line 133
    }
}
```

Make this return the classic \Stripe\Subscription::retrieve($stripeSubscriptionId):

136 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 126
    /**
     * @param $stripeSubscriptionId
     * @return \Stripe\Subscription
     */
    public function findSubscription($stripeSubscriptionId)
    {
        return \Stripe\Subscription::retrieve($stripeSubscriptionId);
    }
}
```

Cool! Back in the controller, add $stripeSubscription = $this->get('stripe_client')->findSubscription() and pass it $stripeSubscriptionId:

79 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 31
        switch ($stripeEvent->type) {
... lines 33 - 39
            case 'invoice.payment_succeeded':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;

                if ($stripeSubscriptionId) {
                    $subscription = $this->findSubscription($stripeSubscriptionId);
                    $stripeSubscription = $this->get('stripe_client')
                        ->findSubscription($stripeSubscriptionId);
... lines 47 - 48
                }
                break;
            default:
                // allow this - we'll have Stripe send us everything
                // throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 55 - 77
}
```

Updating the Subscription in the Database

Finally, let's update the Subscription in our database by using the data on the Stripe subscription object. As usual, we'll add this logic to SubscriptionHelper so we can reuse it later.

Add a new public function called handleSubscriptionPaid() that has two arguments: the Subscription object that just got paid and the related \Stripe\Subscription object that holds the updated details:

90 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 80
    public function handleSubscriptionPaid(Subscription $subscription, \Stripe\Subscription $stripeSubscription)
    {
... lines 83 - 87
    }
}
```

Then, we need to read the current_period_end field. But wait! We totally did this earlier in addSubscriptionToUser(). Steal that line! Paste it here, but rename the variable to $newPeriodEnd:

90 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 80
    public function handleSubscriptionPaid(Subscription $subscription, \Stripe\Subscription $stripeSubscription)
    {
        $newPeriodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);
... lines 84 - 87
    }
}
```

Now, set this billingPeriodEndsAt field via $subscription->setBillingPeriodEnds():

90 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 80
    public function handleSubscriptionPaid(Subscription $subscription, \Stripe\Subscription $stripeSubscription)
    {
        $newPeriodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);

        $subscription->setBillingPeriodEndsAt($newPeriodEnd);
... lines 86 - 87
    }
}
```

But wait! Where's my auto-completion! Oh, that method doesn't exist yet. In Subscription, I'll use the "Code"->"Generate" shortcut to select "Setters" and generate this setter:

136 lines | src/AppBundle/Entity/Subscription.php

```
... lines 1 - 10
class Subscription
{
... lines 13 - 130
    public function setBillingPeriodEndsAt($billingPeriodEndsAt)
    {
        $this->billingPeriodEndsAt = $billingPeriodEndsAt;
    }
}
```

Whoops, then update the method in SubscriptionHelper to be setBillingPeriodEndsAt():

90 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 80
    public function handleSubscriptionPaid(Subscription $subscription, \Stripe\Subscription $stripeSubscription)
    {
        $newPeriodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);

        $subscription->setBillingPeriodEndsAt($newPeriodEnd);
... lines 86 - 87
    }
}
```

Finally, celebrate! Persist and flush the Subscription changes to the database:

90 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 80
    public function handleSubscriptionPaid(Subscription $subscription, \Stripe\Subscription $stripeSubscription)
    {
        $newPeriodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);

        $subscription->setBillingPeriodEndsAt($newPeriodEnd);
        $this->em->persist($subscription);
        $this->em->flush($subscription);
    }
}
```

Back in your controller, call this: $subscriptionHelper->handleSubscriptionPaid() and pass it $subscription and $stripeSubscription:

79 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 39
            case 'invoice.payment_succeeded':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;

                if ($stripeSubscriptionId) {
                    $subscription = $this->findSubscription($stripeSubscriptionId);
                    $stripeSubscription = $this->get('stripe_client')
                        ->findSubscription($stripeSubscriptionId);

                    $subscriptionHelper->handleSubscriptionPaid($subscription, $stripeSubscription);
                }
                break;
            default:
                // allow this - we'll have Stripe send us everything
                // throw new \Exception('Unexpected webhook type form Stripe! '.$stripeEvent->type);
        }
... lines 55 - 56
    }
... lines 58 - 77
}
```

I won't test this - let's call that homework for you - but now whenever a subscription is renewed, the $billingPeriodEndsAt will be updated.

Sending an Email on Renewal

But there's just *one* other small important thing you'll want to do each time a subscription is renewed: send the user an email! Ok, we're not *actually* going to code up the email-sending logic now - but it would live right here in SubscriptionHelper.

But wait! There is one gotcha: the invoice.payment_succeeded webhook will be triggered when a subscription is renewed... but it will *also* be triggered at the moment that the user *originally* buys their subscription. So if you send your user an email that says: "thanks for renewing your subscription", then any new users will be pretty confused.

To fix this, add a new $isRenewal variable set to $newPeriodEnd > $subscription->getBillingPeriodEndsAt():

93 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 80
    public function handleSubscriptionPaid(Subscription $subscription, \Stripe\Subscription $stripeSubscription)
    {
        $newPeriodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);

        // you can use this to send emails to new or renewal customers
        $isRenewal = $newPeriodEnd > $subscription->getBillingPeriodEndsAt();

        $subscription->setBillingPeriodEndsAt($newPeriodEnd);
        $this->em->persist($subscription);
        $this->em->flush($subscription);
    }
}
```

If this is a new subscription, then it was completed about 2 seconds ago, and we would have already set the billingPeriodEndsAt to the correct date. When the webhook fires, the dates will already match. But if this is a renewal, the billingPeriodEndsAt in the database will be for *last* month, and $newPeriodEnd will be for next month.

In other words, you can use the $isRenewal flag to send the right *type* of email.

Chapter 21: Webhook: Payment Failed!

Ok, there's just *one* more webhook we need to worry about, and it's the easiest one: invoice.payment_failed. Send a test webhook for this event.

Refresh RequestBin to check it out.

This webhook type is important for only one reason: to send your user an email so that they know we're having problems charging their card. That's it! We're already using a different webhook to actually *cancel* their subscription if the failures continue.

This has almost the same body as the invoice.payment_succeeded event: the embedded object is an invoice and if that invoice is related to a subscription, it has a subscription property.

That means that in WebhookController, this is a pretty easy one to handle. Add a new case for invoice.payment_failed:

92 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
9   class WebhookController extends BaseController
10  {
... lines 11 - 13
14      public function stripeWebhookAction(Request $request)
15      {
... lines 16 - 31
32          switch ($stripeEvent->type) {
... lines 33 - 50
51              case 'invoice.payment_failed':
... lines 52 - 62
63                  break;
... lines 64 - 66
67          }
... lines 68 - 69
70      }
... lines 71 - 90
91  }
```

Then, start just like before: grab the $stripeSubscriptionId. Then, add an if statement - just in case this invoice has no subscription:

92 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 50
            case 'invoice.payment_failed':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;

                if ($stripeSubscriptionId) {
... lines 55 - 60
                }

                break;
... lines 64 - 66
        }
... lines 68 - 69
    }
... lines 71 - 90
}
```

What to do when a Payment Fails?

Earlier, we talked about what happens when a payment fails. It depends on your Subscription settings in Stripe, but ultimately, Stripe will attempt to charge the card a few times, and then cancel the subscription.

You *could* send your user an email *each* time Stripe tries to charge their card and fails, but that'll probably be a bit annoying. So, I like to send an email *only* after the first attempt fails.

To know if this webhook is being fired after the first, second or third attempt, use a field called attempt_count. If this equals one, send an email. In the controller, add if $stripeEvent->data->object->attempt_count == 1, then send them an email. Well, I'll leave that step to you guys:

92 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 50
            case 'invoice.payment_failed':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;

                if ($stripeSubscriptionId) {
... lines 55 - 56
                    if ($stripeEvent->data->object->attempt_count == 1) {
... line 58
                        // todo - send the user an email about the problem
                    }
                }

                break;
... lines 64 - 66
        }
... lines 68 - 69
    }
... lines 71 - 90
}
```

If you need to know *which* user the subscription belongs to, first fetch the Subscription from the database by using our findSubscription() method. Then, add $user = $subscription->getUser():

92 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
... lines 16 - 31
        switch ($stripeEvent->type) {
... lines 33 - 50
            case 'invoice.payment_failed':
                $stripeSubscriptionId = $stripeEvent->data->object->subscription;

                if ($stripeSubscriptionId) {
                    $subscription = $this->findSubscription($stripeSubscriptionId);

                    if ($stripeEvent->data->object->attempt_count == 1) {
                        $user = $subscription->getUser();
                        // todo - send the user an email about the problem
                    }
                }

                break;
... lines 64 - 66
        }
... lines 68 - 69
    }
... lines 71 - 90
}
```

I like this webhook - it's easy! And actually, we're done with webhooks! Except for preventing replay attacks... which is important, but painless.

Chapter 22: Webhooks: Preventing Replay Attacks

There's one last teeny, tiny little detail we need to worry about with webhooks: replay attacks. These are a security concern but *also* a practical one.

We already know that nobody can send us, random, fake event data because we fetch a fresh event from Stripe:

92 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 8
class WebhookController extends BaseController
{
... lines 11 - 13
    public function stripeWebhookAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);
        if ($data === null) {
            throw new \Exception('Bad JSON body from Stripe!');
        }

        $eventId = $data['id'];

        if ($this->getParameter('verify_stripe_event')) {
            $stripeEvent = $this->get('stripe_client')
                ->findEvent($eventId);
... lines 26 - 28
        }
... lines 30 - 69
    }
... lines 71 - 90
}
```

But, someone *could* intercept a real webhook, and send it to us multiple times. I don't know why they would do that, but weird things would happen.

And there's also the practical concern. Suppose Stripe sends us a webhook and we process it. But somehow, there was a connection problem between our server and Stripe, so Stripe never received our 200 status code. Then, thinking that the webhook failed, Stripe tries to send the webhook again. If this were for an invoice.payment_succeeded event, one user might get *two* subscription renewal emails. That's weird.

Creating the stripe_event_log Table

Let's prevent that. And it's simple: create a database table that records all the event ID's we've handled. Then, query that table before processing a webhook to make sure we haven't seen it before.

In the AppBundle/Entity directory, create a new PHP Class called StripeEventLog:

36 lines | src/AppBundle/Entity/StripeEventLog.php

```
... lines 1 - 2
namespace AppBundle\Entity;
... lines 4 - 10
class StripeEventLog
{
... lines 13 - 34
}
```

Give it a few properties: $id, $stripeEventId and a $handledAt date field:

36 lines | src/AppBundle/Entity/StripeEventLog.php

```
... lines 1 - 10
class StripeEventLog
{
... lines 13 - 17
    private $id;
... lines 19 - 22
    private $stripeEventId;
... lines 24 - 27
    private $handledAt;
... lines 29 - 34
}
```

Since this project uses Doctrine, I'll add a special use statement on top and then add some annotations, so that this new class will become a new table in the database. Use the "Code"->"Generate" menu, or Command + N on a Mac and select "ORM Class":

36 lines | src/AppBundle/Entity/StripeEventLog.php

```
... lines 1 - 4
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="stripe_event_log")
 */
class StripeEventLog
{
... lines 13 - 34
}
```

Repeat that and select "ORM Annotations". Choose all the fields:

36 lines | src/AppBundle/Entity/StripeEventLog.php

```
... lines 1 - 10
class StripeEventLog
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $stripeEventId;

    /**
     * @ORM\Column(type="datetime")
     */
    private $handledAt;
... lines 29 - 34
}
```

Update stripeEventId to be a string field - that'll translate to a varchar in MySQL:

36 lines | src/AppBundle/Entity/StripeEventLog.php

```
... lines 1 - 10
class StripeEventLog
{
... lines 13 - 19
    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $stripeEventId;
... lines 24 - 34
}
```

To set the properties, create a new __construct() method with a $stripeEventId argument. Inside, set that on the property and also set $this->handledAt to a new \DateTime() to set this field to "right now":

36 lines | src/AppBundle/Entity/StripeEventLog.php

```
... lines 1 - 10
class StripeEventLog
{
... lines 13 - 29
    public function __construct($stripeEventId)
    {
        $this->stripeEventId = $stripeEventId;
        $this->handledAt = new \DateTime();
    }
}
```

Brilliant! And now that we have the entity class, find your terminal and run:

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

This generates a new file in the app/DoctrineMigrations directory that contains the raw SQL needed to create the new table:

35 lines | app/DoctrineMigrations/Version20160807113428.php

```
... lines 1 - 2
namespace Application\Migrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20160807113428 extends AbstractMigration
{
    /**
     * @param Schema $schema
     */
    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('CREATE TABLE stripe_event_log (id INT AUTO_INCREMENT NOT NULL, stripe_event_id VARCHAR(255) NOT
    }

    /**
     * @param Schema $schema
     */
    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 TABLE stripe_event_log');
    }
}
```

Execute that query by running:

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

Preventing the Replay Attack

Finally, in WebhookController, start by querying to see if this event has been handled before. Fetch the EntityManager, and then add $existingLog = $em->getRepository('AppBundle:StripeEventLog') and call findOneBy() on it to query for stripeEventId set to $eventId.

104 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 9
class WebhookController extends BaseController
{
... lines 12 - 14
    public function stripeWebhookAction(Request $request)
    {
... lines 17 - 21
        $eventId = $data['id'];

        $em = $this->getDoctrine()->getManager();
        $existingLog = $em->getRepository('AppBundle:StripeEventLog')
            ->findOneBy(['stripeEventId' => $eventId]);
... lines 27 - 81
    }
... lines 83 - 102
}
```

If an $existingLog is found, then we don't want to handle this. Just return a new Response() that says "Event previously handled":

104 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 9
class WebhookController extends BaseController
{
... lines 12 - 14
    public function stripeWebhookAction(Request $request)
    {
... lines 17 - 21
        $eventId = $data['id'];

        $em = $this->getDoctrine()->getManager();
        $existingLog = $em->getRepository('AppBundle:StripeEventLog')
            ->findOneBy(['stripeEventId' => $eventId]);
        if ($existingLog) {
            return new Response('Event previously handled');
        }
... lines 30 - 81
    }
... lines 83 - 102
}
```

If you also want to log a message so that you know when this happens, that's not a bad idea.

But if there is *not* an existing log, time to process this webhook! Create a new StripeEventLog and pass it $eventId. Then, persist and flush *just* the log:

104 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 9
class WebhookController extends BaseController
{
... lines 12 - 14
    public function stripeWebhookAction(Request $request)
    {
... lines 17 - 21
        $eventId = $data['id'];

        $em = $this->getDoctrine()->getManager();
        $existingLog = $em->getRepository('AppBundle:StripeEventLog')
            ->findOneBy(['stripeEventId' => $eventId]);
        if ($existingLog) {
            return new Response('Event previously handled');
        }

        $log = new StripeEventLog($eventId);
        $em->persist($log);
        $em->flush($log);
... lines 34 - 81
    }
... lines 83 - 102
}
```

And yea, replay attacks are gone!

Update the Test!

To make sure we didn't mess anything up, open WebhookControllerTest and copy our test method. Run that:

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

Bah! Of course... it failed for a silly reason: I need to update my *test* database - to add the new table. A shortcut to do that is:

```
$ ./bin/console doctrine:schema:update --force --env=test
```

Try the test now:

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

It works! So hey, run it again!

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

It fails?!

> Failed to assert that true is false.

Well, that's not clear, but I know what the problem is: every event in the test has the *same* event ID:

128 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 70
    private function getCustomerSubscriptionDeletedEvent($subscriptionId)
    {
        $json = <<<EOF
{
... lines 75 - 76
  "id": "evt_00000000000000",
... lines 78 - 121
}
EOF;
... lines 124 - 125
    }
}
```

So when you run the test the second time, this already exists in the StripeEventLog table and the webhook is skipped. Well hey, at least we know the replay attack system is working.

To fix this, we need to set a little bit of randomness to the event ID by adding a %s at the end and adding an mt_rand() to the sprintf():

128 lines | tests/AppBundle/Controller/WebhookControllerTest.php

```
... lines 1 - 9
class WebhookControllerTest extends WebTestCase
{
... lines 12 - 70
    private function getCustomerSubscriptionDeletedEvent($subscriptionId)
    {
        $json = <<<EOF
{
... lines 75 - 76
  "id": "evt_00000000000000%s",
... lines 78 - 121
}
EOF;

        return sprintf($json, mt_rand(), $subscriptionId);
    }
}
```

Now, every event ID will be unique. Try the test again:

```
$ ./vendor/bin/phpunit --filter testStripeCustomerSubscriptionDeleted
```

Green and happy!

Ok, *enough* webhooks. Let's do something fun, like making it possible for a user to *upgrade* from one subscription to another.

Chapter 23: Upgrading Subscription Plans: The UI

Imagine this: a sheep customer is *so* happy with the Farmer Brent subscription that they want to *upgrade* to the New Zealander! Awesome! Amazing! But also... currently impossible.

Time to fix that. Plan upgrades can be complex, because someone needs to calculate how much of the current month has been used and prorate funds towards the upgrade. Stripe's documentation talks about this. I'll guide you through everything, but this section is worth a read.

Printing the Current Plan

Buy a new Farmer Brent Subscription, and then head to the account page.

Let's focus on the plan upgrade user interface first. Here, I need to see *which* plan I'm currently subscribed to and a button to upgrade to the other plan.

Open the account.html.twig template and ProfileController.

Add a new variable: $currentPlan = null. Then, only *if* $this->getUser()->hasActiveSubscription(), set $currentPlan = $this->get('subscription_helper')->findPlan() passing that $this->getUser()->getSubscription()->getStripePlanId():

108 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 17
    public function accountAction()
    {
        $currentPlan = null;
        if ($this->getUser()->hasActiveSubscription()) {
            $currentPlan = $this->get('subscription_helper')
                ->findPlan($this->getUser()->getSubscription()->getStripePlanId());
        }
... lines 25 - 30
    }
... lines 32 - 106
}
```

The findPlan() method will give *us* a fancy SubscriptionPlan object.

Pass a new currentPlan variable into the template, set to $currentPlan:

108 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 17
    public function accountAction()
    {
        $currentPlan = null;
        if ($this->getUser()->hasActiveSubscription()) {
            $currentPlan = $this->get('subscription_helper')
                ->findPlan($this->getUser()->getSubscription()->getStripePlanId());
        }

        return $this->render('profile/account.html.twig', [
... lines 27 - 28
            'currentPlan' => $currentPlan
        ]);
    }
... lines 32 - 106
}
```

Then in the template, find the "Active Subscription" spot, and print currentPlan.name:

101 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 18
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 24 - 39
                <table class="table">
                    <tbody>
                        <tr>
                            <th>Subscription</th>
                            <td>
                                {% if app.user.hasActiveSubscription %}
                                    {% if app.user.subscription.isCancelled %}
... lines 47 - 49
                                    {% else %}
                                        {{ currentPlan.name }}

                                        <span class="label label-success">Active</span>
                                    {% endif %}
... lines 55 - 56
                                {% endif %}
                            </td>
                        </tr>
... lines 60 - 83
                    </tbody>
                </table>
            </div>
... lines 87 - 95
        </div>
    </div>
</div>
{% endblock %}
... lines 100 - 101
```

Refresh the page! Great! Step 1 done: we have the "Farmer Brent" plan.

Adding the Upgrade Button

Now, step two: add an upgrade button that mentions the plan they could switch to. Since we only have 2 plans, it's pretty simple: if they're on the Farmer Brent, we want to allow them to upgrade to the New Zealander. And if they're on the New Zealander, we should let them *downgrade* to the Farmer Brent.

To find the *other* plan, open SubscriptionHelper and add a new public function called findPlanToChangeTo with a $currentPlanId argument:

108 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 45
    /**
     * @param $currentPlanId
     * @return SubscriptionPlan
     */
    public function findPlanToChangeTo($currentPlanId)
    {
        if (strpos($currentPlanId, 'farmer_brent') !== false) {
            $newPlanId = str_replace('farmer_brent', 'new_zealander', $currentPlanId);
        } else {
            $newPlanId = str_replace('new_zealander', 'farmer_brent', $currentPlanId);
        }

        return $this->findPlan($newPlanId);
    }
... lines 60 - 106
}
```

I'll paste in the logic: it's kind of silly, but it gets the job done. I'm using str_replace instead of something simpler, because in a few minutes, we're going to add *yearly* plans, and I still want this function to... um... function.

Back to the controller! Add another variable: $otherPlan = null. Then, $otherPlan = $this->get('subscription_helper')->findPlanToChangeTo() and pass it $currentPlan->getPlanId(). Pass this into the template as an otherPlan variable:

113 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 12
class ProfileController extends BaseController
{
... lines 15 - 17
    public function accountAction()
    {
        $currentPlan = null;
        $otherPlan = null;
        if ($this->getUser()->hasActiveSubscription()) {
            $currentPlan = $this->get('subscription_helper')
                ->findPlan($this->getUser()->getSubscription()->getStripePlanId());

            $otherPlan = $this->get('subscription_helper')
                ->findPlanToChangeTo($currentPlan->getPlanId());
        }

        return $this->render('profile/account.html.twig', [
... lines 31 - 32
            'currentPlan' => $currentPlan,
            'otherPlan' => $otherPlan,
        ]);
    }
... lines 37 - 111
}
```

There, after the "Active" label, add a button with some classes: a few for styling, one to float right and one -

js-change-plan-button - that we'll use in a minute via JavaScript. Make the text: "Change to" and then otherPlan.name:

111 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 24
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 30 - 45
                <table class="table">
                    <tbody>
                        <tr>
                            <th>Subscription</th>
                            <td>
                                {% if app.user.hasActiveSubscription %}
                                    {% if app.user.subscription.isCancelled %}
... lines 53 - 55
                                    {% else %}
                                        {{ currentPlan.name }}

                                        <span class="label label-success">Active</span>

                                        <button class="btn btn-xs btn-link pull-right js-change-plan-button" data-plan-id="{{ otherPlan.planId }}" data
                                            Change to {{ otherPlan.name }}
                                        </button>
                                    {% endif %}
... lines 65 - 66
                                {% endif %}
                            </td>
                        </tr>
... lines 70 - 93
                    </tbody>
                </table>
            </div>
... lines 97 - 105
        </div>
    </div>
</div>
{% endblock %}
... lines 110 - 111
```

Oh, and add one more attribute: data-plan-name and print otherPlan.name. We'll read that attribute in JavaScript.

Bootstrapping the JavaScript

In fact, let's play with the JavaScript right now: copy the js-change-plan-button class and find the JavaScript block at the top of this file. Use jQuery to locate that element, then on click, add a callback. Start with the always-in-style e.preventDefault():

111 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
                e.preventDefault();
... lines 18 - 19
            })
        });
    </script>
{% endblock %}
... lines 24 - 111
```

Start really simple: we'll use a library that I already installed called Sweet Alerts. Call swal() and pass a message Loading Plan Details:

111 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
                e.preventDefault();

                swal('Loading Plan Details...');
            })
        });
    </script>
{% endblock %}
... lines 24 - 111
```

Ok, let's see what this Sweet Alerts thing looks like! Refresh that page! Nice! Click the "Change to New Zealander" link. This is Sweet Alert. It's cute, it's easy, and it'll help us do our job.

Because next, we need to do some serious work: we need to calculate how *much* we should charge the user to upgrade from the Farmer Brent to the New Zealander, and then show it to the user. That's tricky, because the user is probably in the middle of the month that they've already paid for, so they deserve some credits!

Thankfully, Stripe is going to be a *champ* and help us out.

Chapter 24: So, how much would that Upgrade Cost?

Honestly, upgrade and downgrading a plan would be really easy, except that we need to calculate how much we should charge the user and *tell* them so they can confirm.

Getting this right takes some work, but the result is going to be *gorgeous*, I promise. Here's the plan: as soon as this screen loads, we'll make an AJAX call back to the server. The server will calculate how much to charge the customer for the upgrade, and send that back so we can show it.

In ProfileController, add a new public function called previewPlanChangeAction(). Set the URL to /profile/plan/change/preview/{planId} and give it a name: account_preview_plan_change:

133 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 113
    /**
     * @Route("/profile/plan/change/preview/{planId}", name="account_preview_plan_change")
     */
    public function previewPlanChangeAction($planId)
    {
... lines 119 - 130
    }
}
```

Use the $planId in the route to load a $plan object with $this->get('subscription_helper') and then call findPlan() with $planId:

133 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 116
    public function previewPlanChangeAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);
... lines 121 - 130
    }
}
```

Upcoming Invoice to the Rescue

Ok ok, but I'm ignoring the big, *huge* elephant in the room: how the heck are we going to figure out how much to charge the user? I mean, I certainly don't want to try to calculate how *far* through the month the user is and figure out a prorated amount. Fortunately, we don't have to: Stripe has a killer feature to help us out.

Open the Stripe Api docs and find Invoices. Check out this "Upcoming Invoices" section. Cool. With upcoming invoices, we can ask Stripe to tell us what the Customer's *next* invoice will look like.

This could be used to show the user how much they'll be charged on renewal, *or*, by passing a subscription_plan parameter, this will return an Invoice that describes how much they would be charged for *changing* to that plan.

How Upgrades Work

A big part of all of this is *prorating*. In the subscription documentation, Stripe talks a lot about what will happen in different scenarios. By default, Stripe *does* prorate, which means that if we are 1/4th through the month on the Farmer Brent Plan and we upgrade, then 3/4th's of that cost should be credited as a discount towards paying for the final 3/4th's of a month of the New Zealander plan. When you switch between plans that have the same *duration*, like a monthly plan to another monthly plan, the billing period doesn't change: you simply switch to the new plan right in the middle of the month, and are billed normally again on your normal billing date.

Yea, it's hard! The tl;dr is that Stripe does these calculations for us.

Fetching the Upcoming Invoice

Let's use this endpoint: in StripeClient, add a new function: getUpcomingInvoiceForChangedSubscription() with two arguments: the User that will be upgrading and the SubscriptionPlan they want to change to:

145 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 135
    public function getUpcomingInvoiceForChangedSubscription(User $user, SubscriptionPlan $newPlan)
    {
... lines 138 - 142
    }
}
```

Inside, it's easy: return \Stripe\Invoice::upcoming() and pass it a few parameters. First, customer set to $user->getStripeCustomerId() and second, subscription set to $user->getSubscription()->getStripeSubscriptionId():

145 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 135
    public function getUpcomingInvoiceForChangedSubscription(User $user, SubscriptionPlan $newPlan)
    {
        return \Stripe\Invoice::upcoming([
            'customer' => $user->getStripeCustomerId(),
            'subscription' => $user->getSubscription()->getStripeSubscriptionId(),
... line 141
        ]);
    }
}
```

This tells Stripe *which* subscription we would update. Now, in our system, every user should only have one, but it doesn't hurt to be explicit.

The last option is subscription_plan: in other words, which plan do we want to change to. Set it to $newPlan->getPlanId():

145 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 135
    public function getUpcomingInvoiceForChangedSubscription(User $user, SubscriptionPlan $newPlan)
    {
        return \Stripe\Invoice::upcoming([
            'customer' => $user->getStripeCustomerId(),
            'subscription' => $user->getSubscription()->getStripeSubscriptionId(),
            'subscription_plan' => $newPlan->getPlanId(),
        ]);
    }
}
```

Back in ProfileController, use this to set a new $stripeInvoice variable via this->get('stripe_client')->getUpcomingInvoiceForChangedSubscription() passing it $this->getUser() and the new $plan:

133 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 116
    public function previewPlanChangeAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );
... lines 127 - 130
    }
}
```

Dumping the Upcoming Invoice

So, what does this fancy Upcoming invoice actually look like? Let's find out by dumping it. Then, return a JsonResponse with... I don't know, how about a total key set to a hardcoded 50 for now. Oh, and make sure you dump $stripeInvoice:

133 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 116
    public function previewPlanChangeAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );

        dump($stripeInvoice);

        return new JsonResponse(['total' => 50]);
    }
}
```

Ok, let's keep going by hooking up the frontend and finishing the cost calculation.

Chapter 25: Upgrade: Processing the Upcoming Invoice

Head over to the account template. When the user clicks upgrade, we need to make an AJAX call to our new endpoint. To get that URL, find the button and add a new attribute: data-preview-url set to path('account_preview_plan_change'), passing a planId wildcard set to otherPlan.planId:

123 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 33
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 39 - 54
                <table class="table">
                    <tbody>
                        <tr>
                            <th>Subscription</th>
                            <td>
                                {% if app.user.hasActiveSubscription %}
                                    {% if app.user.subscription.isCancelled %}
... lines 62 - 64
                                    {% else %}
... lines 66 - 69
                                        <button class="btn btn-xs btn-link pull-right js-change-plan-button"
                                            data-preview-url="{{ path('account_preview_plan_change', {'planId': otherPlan.planId}) }}"
                                            data-plan-name="{{ otherPlan.name }}"
                                        >
                                            Change to {{ otherPlan.name }}
                                        </button>
                                    {% endif %}
... lines 77 - 78
                                {% endif %}
                            </td>
                        </tr>
... lines 82 - 105
                    </tbody>
                </table>
            </div>
... lines 109 - 117
        </div>
    </div>
</div>
{% endblock %}
... lines 122 - 123
```

Cool! Copy that new attribute name and go back up to the JavaScript section. Let's read that attribute: var previewUrl = $(this).data('preview-url'). And while we're here, create a planName variable set to $(this).data('plan-name'):

123 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
                e.preventDefault();

                swal('Loading Plan Details...');

                var previewUrl = $(this).data('preview-url');
                var planName = $(this).data('plan-name');
... lines 23 - 28
            })
        });
    </script>
{% endblock %}
... lines 33 - 123
```

Now, make that AJAX call! I'll use $.ajax() with url set to previewUrl. Chain a .done() to add a success function with a data argument. And *just* to try things out, open sweet alert with a message: Total $ then data.total, since the endpoint returns that field:

123 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
                e.preventDefault();

                swal('Loading Plan Details...');

                var previewUrl = $(this).data('preview-url');
                var planName = $(this).data('plan-name');

                $.ajax({
                    url: previewUrl
                }).done(function(data) {
                    swal('Total $'+data.total);
                });
            })
        });
    </script>
{% endblock %}
... lines 33 - 123
```

Ok team, try that out. Refresh the account page and click "Change to New Zealander". Bam! Total $50!

Using the Upcoming Invoice

With the frontend *somewhat* functional, let's finish the logic in our endpoint. At the bottom, Symfony keeps a list of the AJAX requests. Click the 4f4 sha link to get more information about our AJAX request. Then, click the Debug link on the left.

In the last chapter, we dumped the upcoming \Stripe\Invoice object that we got from the Stripe API. This is it! It looks a little funny, but the data is hiding under the _values property, and it holds a couple of *really* interesting things.

Upcoming Invoice Line Items

First, check out amount_due, and remember, everything is stored in *cents*, not dollars. This is the amount we'll show to the user. But if it seems a little too high, you're right. Keep watching.

Second, the invoice line items can be found under the lines key. And there are *three*.

The first line item is *negative*: its a credit for any unused time on your current plan. If you're half-way through a month, then the second half should be applied as a credit. This is that credit. Since we just signed up a few minutes ago, this is just *slightly* less than the full price of $99.

The second line item is a charge for the new plan, for however much time is left in the month. Again, if we're upgrading half-way through the month, I should only need to pay for *half* of the new plan in order to use it for the last *half* of the month.

The third line item, well, this is where things get ugly. This is a charge for a *full* month on the new plan: $199.

What? Why is that here? Why would I pay for half of the month of the New Zealander plan and *also* for a full month?

Here's what's going on: when a customer upgrades, Stripe does *not* charge them *anything* immediately. Instead, Stripe allows you to switch, but then, at the end of the month, it will charge you for the partial, prorated month you just used, plus the full *next* month, minus the partial-month refund for your original plan.

Phew! That's why you see three line items: the first two for adjusting to the new plan for part of the month, plus the cost for the full-price renewal.

Charging Immediately for an Upgrade

Honestly, this feels weird to me. So let's do something better: let's charge the customer *immediately* for the plan price change, and then let them pay for the normal, full-month renewal next month. This is totally possible to do.

But that means, to show the user the amount they will be charged right now, we need to read the amount_due value and then *subtract* the full price of the plan, to remove the extra line item.

In ProfileController, add a new variable $total set to $stripeInvoice->amount_due:

137 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 116
    public function previewPlanChangeAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );

        // contains the pro-rations *plus* the next cycle's amount
        $total = $stripeInvoice->amount_due;
... lines 130 - 134
    }
}
```

Add a comment above - this stuff is confusing, so let's leave some notes. Then, correct the total by subtracting $plan->getPrice() * 100 to convert into cents - our price is stored in dollars:

137 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 116
    public function previewPlanChangeAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );

        // contains the pro-rations *plus* the next cycle's amount
        $total = $stripeInvoice->amount_due;

        // subtract plan price to *remove* next the next cycle's total
        $total -= $plan->getPrice() * 100;
... lines 133 - 134
    }
}
```

Then, return $total / 100 in the JSON:

137 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 13
class ProfileController extends BaseController
{
... lines 16 - 116
    public function previewPlanChangeAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );

        // contains the pro-rations *plus* the next cycle's amount
        $total = $stripeInvoice->amount_due;

        // subtract plan price to *remove* next the next cycle's total
        $total -= $plan->getPrice() * 100;

        return new JsonResponse(['total' => $total/100]);
    }
}
```

Let's try it guys: go back and refresh.

Click "Change to New Zealander". Ok, $99.93 - that *looks* about right. Remember, the upgrade should cost about $100, but

since we've been using the old plan for a few minutes, the true cost should be *slightly* lower.

Finishing up the JS

Ok! It's time to *execute* this upgrade! To save us some time, I'll paste some JavaScript into the AJAX success function:

139 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
... lines 17 - 23
                $.ajax({
                    url: previewUrl
                }).done(function(data) {
                    var message;
                    if (data.total > 0) {
                        message = 'You will be charged $'+data.total +' immediately';
                    } else {
                        message = 'You will have a balance of $'+(Math.abs(data.total))+' that will be automatically applied to future invoices!';
                    }

                    swal({
                        title: 'Change to '+planName,
                        text: message,
                        type: "info",
                        showCancelButton: true,
                        closeOnConfirm: false,
                        showLoaderOnConfirm: true
                    }, function () {
                        // todo - actually change the plan!
                    });
                });
            })
        });
    </script>
{% endblock %}
... lines 49 - 139
```

This first display how much we will charge the user. And check this out: it could be *positive*, meaning we'll charge them, or *negative* for a downgrade, meaning they'll get an account credit that will automatically be used for future charges.

Finally, this shows the user *one* last alert to confirm the change. If they click "Ok", the last callback will be executed. And it'll be our job to send one more AJAX call back to the server to finally change their plan.

Let's do it!

Chapter 26: Execute the Plan Upgrade

When the user clicks "OK", we'll make an AJAX request to the server and then tell Stripe to *actually* make the change.

In ProfileController, add the new endpoint: public function changePlanAction(). Set its URL to /profile/plan/change/execute/{planId} and name it account_execute_plan_change. Add the $planId argument:

157 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 137
    /**
     * @Route("/profile/plan/change/execute/{planId}", name="account_execute_plan_change")
     * @Method("POST")
     */
    public function changePlanAction($planId)
    {
... lines 144 - 154
    }
}
```

This will start just like the previewPlanChangeAction() endpoint: copy its $plan code and paste it here:

157 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 137
    /**
     * @Route("/profile/plan/change/execute/{planId}", name="account_execute_plan_change")
     * @Method("POST")
     */
    public function changePlanAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeClient = $this->get('stripe_client');
        $stripeSubscription = $stripeClient->changePlan($this->getUser(), $plan);

        // causes the planId to be updated on the user's subscription
        $this->get('subscription_helper')
            ->addSubscriptionToUser($stripeSubscription, $this->getUser());

        return new Response(null, 204);
    }
}
```

Changing a Subscription Plan in Stripe

To actually change the plan in Stripe, we need to fetch the *Subscription*, set its plan to the new id, and save. Super easy!

Open StripeClient and add a new function called changePlan() with two arguments: the User who wants to upgrade and the SubscriptionPlan that they want to change to:

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 144
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 147 - 155
    }
}
```

Then, fetch the \Stripe\Subscription for the User with $this->findSubscription() passing it $user->getSubscription()->getStripeSubscriptionId():

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 144
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());
... lines 148 - 155
    }
}
```

Now, update that: $stripeSubscription->plan = $newPlan->getPlanId():

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 146
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());

        $stripeSubscription->plan = $newPlan->getPlanId();
... lines 150 - 158
```

Finally, send that to Stripe with $stripeSubscription->save():

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 146
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());

        $stripeSubscription->plan = $newPlan->getPlanId();
        $stripeSubscription->save();
... lines 151 - 158
```

But Charge the User Immediately

Ok, that *was* easy. And now you probably expect there to be a "catch" or a gotcha that makes this harder. Well... yea... there totally is. Sorry.

I told you earlier that Stripe doesn't charge the customer right now: it waits until the end of the cycle and then bills for next month's renewal, plus what they owe for upgrading this month. We want to bill them immediately.

How? Simple: by manually creating an Invoice and paying it. Remember: when you create an Invoice, Stripe looks for all unpaid invoice items on the customer. When you change the plan, this creates *two* new invoice items for the negative and positive plan proration. So if we invoice the user right now, it will pay those invoice items.

And hey! We *already* have a method to do that called createInvoice(). Heck it even *pays* that invoice immediately:

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 53
    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;
    }
... lines 67 - 156
}
```

In our function, call $this->createInvoice() and pass it $user:

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 144
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());

        $stripeSubscription->plan = $newPlan->getPlanId();
        $stripeSubscription->save();

        // immediately invoice them
        $this->createInvoice($user);
... lines 154 - 155
    }
}
```

Finally, return $stripeSubscription at the bottom - we'll need that in a minute:

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 144
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());

        $stripeSubscription->plan = $newPlan->getPlanId();
        $stripeSubscription->save();

        // immediately invoice them
        $this->createInvoice($user);

        return $stripeSubscription;
    }
}
```

Back in the controller, call this with $stripeSubscription = $this->get('stripe_client') then ->changePlan($this->getUser(), $plan):

157 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 141
    public function changePlanAction($planId)
    {
        $plan = $this->get('subscription_helper')
            ->findPlan($planId);

        $stripeClient = $this->get('stripe_client');
        $stripeSubscription = $stripeClient->changePlan($this->getUser(), $plan);
... lines 149 - 154
    }
}
```

Upgrading the Plan in our Database

Ok, the plan is upgraded! Well, in Stripe. But we *also* need to update the subscription row in our database.

When a user buys a new subscription, we call a method on SubscriptionHelper called addSubscriptionToUser(). We pass it the new \Stripe\Subscription and the User:

108 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 60
    public function addSubscriptionToUser(\Stripe\Subscription $stripeSubscription, User $user)
    {
        $subscription = $user->getSubscription();
        if (!$subscription) {
            $subscription = new Subscription();
            $subscription->setUser($user);
        }

        $periodEnd = \DateTime::createFromFormat('U', $stripeSubscription->current_period_end);
        $subscription->activateSubscription(
            $stripeSubscription->plan->id,
            $stripeSubscription->id,
            $periodEnd
        );

        $this->em->persist($subscription);
        $this->em->flush($subscription);
    }
... lines 79 - 106
}
```

Then *it* guarantees that the user has a subscription row in the table with the correct data, like the plan id, subscription id, and $periodEnd date.

Now, the only thing *we* need to update right now is the plan ID: both the subscription ID and period end haven't changed. But that's ok: we can still safely reuse this method.

In ProfileController, add $this->get('subscription_helper')->addSubscriptionToUser() passing it $stripeSubscription and $this->getUser():

157 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 141
    public function changePlanAction($planId)
    {
... lines 144 - 147
        $stripeSubscription = $stripeClient->changePlan($this->getUser(), $plan);

        // causes the planId to be updated on the user's subscription
        $this->get('subscription_helper')
            ->addSubscriptionToUser($stripeSubscription, $this->getUser());
... lines 153 - 154
    }
}
```

And that's *everything*. At the bottom... well, we don't *really* need to return anything to our JSON. So just return a new Response() with null as the content and a 204 status code:

157 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 141
    public function changePlanAction($planId)
    {
... lines 144 - 147
        $stripeSubscription = $stripeClient->changePlan($this->getUser(), $plan);

        // causes the planId to be updated on the user's subscription
        $this->get('subscription_helper')
            ->addSubscriptionToUser($stripeSubscription, $this->getUser());

        return new Response(null, 204);
    }
}
```

This doesn't do anything special: 204 simply means that the operation was successful, but the server has nothing it wishes to say back.

Executing the Upgrade in the UI

Copy the route name, then head to the template to make this work.

First, find the button, copy the data-preview-url attribute, and paste it. Name the new one data-change-url and update the route name:

152 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 61
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 67 - 82
                <table class="table">
                    <tbody>
                        <tr>
                            <th>Subscription</th>
                            <td>
                                {% if app.user.hasActiveSubscription %}
                                    {% if app.user.subscription.isCancelled %}
... lines 90 - 92
                                    {% else %}
... lines 94 - 97
                                        <button class="btn btn-xs btn-link pull-right js-change-plan-button"
                                            data-preview-url="{{ path('account_preview_plan_change', {'planId': otherPlan.planId}) }}"
                                            data-plan-name="{{ otherPlan.name }}"
                                            data-change-url="{{ path('account_execute_plan_change', {'planId': otherPlan.planId}) }}"
                                        >
                                            Change to {{ otherPlan.name }}
                                        </button>
                                    {% endif %}
... lines 106 - 107
                                {% endif %}
                            </td>
                        </tr>
... lines 111 - 134
                    </tbody>
                </table>
            </div>
... lines 138 - 146
        </div>
    </div>
</div>
{% endblock %}
... lines 151 - 152
```

Above in the JavaScript, set a new changeUrl variable to $(this).data('change-url'):

152 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
... lines 17 - 20
                var previewUrl = $(this).data('preview-url');
                var changeUrl = $(this).data('change-url');
... lines 23 - 56
            })
        });
    </script>
{% endblock %}
... lines 61 - 152
```

Then, scroll down to the bottom: this callback function will be executed *if* the user clicks the "Ok" button to confirm the change. Make the AJAX call here: set the url to changeUrl, the method to POST, and attach *one* more success function:

152 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
... lines 17 - 24
                $.ajax({
                    url: previewUrl
                }).done(function(data) {
... lines 28 - 34
                    swal({
                        title: 'Change to '+planName,
                        text: message,
                        type: "info",
                        showCancelButton: true,
                        closeOnConfirm: false,
                        showLoaderOnConfirm: true
                    }, function () {
                        $.ajax({
                            url: changeUrl,
                            method: 'POST'
                        }).done(function() {
... lines 47 - 52
                        });
                        // todo - actually change the plan!
                    });
                });
            })
        });
    </script>
{% endblock %}
... lines 61 - 152
```

Inside that, call Sweet Alert to tell the user that the plan was changed! Let's also add some code to reload the page after everything:

152 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
... lines 17 - 24
                $.ajax({
                    url: previewUrl
                }).done(function(data) {
... lines 28 - 34
                    swal({
                        title: 'Change to '+planName,
                        text: message,
                        type: "info",
                        showCancelButton: true,
                        closeOnConfirm: false,
                        showLoaderOnConfirm: true
                    }, function () {
                        $.ajax({
                            url: changeUrl,
                            method: 'POST'
                        }).done(function() {
                            swal({
                                title: 'Plan changed!',
                                type: 'success'
                            }, function() {
                                location.reload();
                            });
                        });
                        // todo - actually change the plan!
                    });
                });
            })
        });
    </script>
{% endblock %}
... lines 61 - 152
```

OK! Let's do this! Refresh the page! Click to change to the "New Zealander". $99.88 - that looks right, now press "Ok". And ... cool! I think it worked! When the page reloads, our plan is the "New Zealander" and we can downgrade to the "Farmer Brent".

In the Stripe dashboard, open payments, click the one for $99.88, and open its Invoice. Oh, it's a thing of beauty: this has the two line items for the change.

If you check out the customer, their top subscription is *now* to the New Zealander plan.

So we're good. Except for one last edge-case.

Chapter 27: Failing Awesomely When Payments Fail

You can even *downgrade* to the Farmer Brent and get a $99 balance on your account. In the Stripe dashboard, refresh the Customer. Ah, there's an account balance of $99.84.

And if you change *back* to the New Zealander, now it's free! The amount_due field on the upcoming invoice correctly calculates its total by using any available account balance.

So, this is solid.

Card Failure on Upgrade

But what happens if their card is declined when they try to upgrade? I don't know: let's find out. First, go buy a new, fresh subscription. Great!

Now, update your card to be one that will fail when it's charged: 4000 0000 0000 0341.

Ok, try to change to the New Zealander. It thinks for awhile and then... an AJAX error! You can see it down in the web debug toolbar.

Open the profiler for that request in a new tab and then click "Exception" to see the error. Ah yes, the classic: "Your card was declined". Clearly, we aren't handling this situation very well.

But actually, the problem is worse than you might think. Refresh the Customer in Stripe. You can see the failed payment... but you can also see that the subscription change was successful! We are now on the New Zealander plan.

The customer also has an unpaid invoice, which represents what they should have been charged. Since this is unpaid, Stripe will try to charge it a few more times. In summary, everything is totally borked.

Failing Gracefully

This whole mess starts in StripeClient, when we call $this->createInvoice(), because this might fail:

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 144
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 147 - 151
        // immediately invoice them
        $this->createInvoice($user);
... lines 154 - 155
    }
}
```

Scroll up to that method:

158 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 8
class StripeClient
{
... lines 11 - 53
    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;
    }
... lines 67 - 156
}
```

In addition to calling this to upgrade a subscription plan, we *also* call this at checkout, even if there is *no* subscription. The problem is that if payment fails on checkout, this invoice will *still* exist, and Stripe will try again to charge it. Imagine having your card be declined at checkout, only for the vendor to try to charge it again later, without you ever having finished the checkout process!

Here's our rescue plan: if paying the invoice fails, we need to *close* it. By doing that, Stripe will *not* try to pay it again.

To do that, surround the pay() line with a try-catch for the \Stripe\Error\Card exception:

177 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 8
class StripeClient
{
... lines 11 - 53
    public function createInvoice(User $user, $payImmediately = true)
    {
... lines 56 - 59
        if ($payImmediately) {
            // guarantee it charges *right* now
            try {
                $invoice->pay();
            } catch (\Stripe\Error\Card $e) {
... lines 65 - 70
            }
        }
... lines 73 - 74
    }
... lines 76 - 175
}
```

Here, add $invoice->close = true and then $invoice->save(). Then, re-throw the exception:

177 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 8
class StripeClient
{
... lines 11 - 53
    public function createInvoice(User $user, $payImmediately = true)
    {
... lines 56 - 59
        if ($payImmediately) {
            // guarantee it charges *right* now
            try {
                $invoice->pay();
            } catch (\Stripe\Error\Card $e) {
                // paying failed, close this invoice so we don't
                // keep trying to pay it
                $invoice->closed = true;
                $invoice->save();

                throw $e;
            }
        }
... lines 73 - 74
    }
... lines 76 - 175
}
```

Our checkout logic looks for this exception and uses it to notify the user of the problem.

Next, down in the other function, if we fail to create the invoice, we need to *not* change the customer's plan in Stripe.

Add a new variable called $originalPlanId set to $stripeSubscription->plan->id:

177 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());

        $originalPlanId = $stripeSubscription->plan->id;
... lines 159 - 174
    }
}
```

Then, surround the createInvoice() call with a try-catch block for the same exception: \Stripe\Error\Card:

177 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 156 - 161
        try {
            // immediately invoice them
            $this->createInvoice($user);
        } catch (\Stripe\Error\Card $e) {
... lines 166 - 171
        }
... lines 173 - 174
    }
}
```

Reverting the Plan without Proration

If this happens, we need to do change the subscription plan *back* to the original one: $stripeSubscription->plan = $originalPlanId. But here's the tricky part: add $stripeSubscription->prorate = false:

177 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 156 - 161
        try {
            // immediately invoice them
            $this->createInvoice($user);
        } catch (\Stripe\Error\Card $e) {
            $stripeSubscription->plan = $originalPlanId;
            // prevent prorations discounts/charges from changing back
            $stripeSubscription->prorate = false;
... lines 169 - 171
        }
... lines 173 - 174
    }
}
```

Why? When we originally change the plan, that creates the two proration invoice items. If the invoice fails to pay, then the invoice containing those invoice items is closed. And that means, effectively, those invoice items have been deleted, which is good! In fact, it's exactly what we want.

But when we change from the new plan back to the *old* plan, we don't want two *new* proration invoice items in reverse to be created. By saying prorate = false, we're telling Stripe to change back to the original plan, but without creating any invoice items. Yep, simply change the plan back.

Finally, call $stripeSubscription->save(). Then, once again, re-throw the exception:

177 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
use AppBundle\Subscription\SubscriptionPlan;
... lines 7 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 156 - 161
        try {
            // immediately invoice them
            $this->createInvoice($user);
        } catch (\Stripe\Error\Card $e) {
            $stripeSubscription->plan = $originalPlanId;
            // prevent prorations discounts/charges from changing back
            $stripeSubscription->prorate = false;
            $stripeSubscription->save();

            throw $e;
        }
... lines 173 - 174
    }
}
```

Telling the User What Happened

That fixes the problem in Stripe. The last thing *we* need to do is tell the user what went wrong.

Open ProfileController::changePlanAction(). Surround the changePlan() call with - you guessed it - one more try-catch block for that same exception: \Stripe\Error\Card:

164 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 141
    public function changePlanAction($planId)
    {
... lines 144 - 148
        try {
            $stripeSubscription = $stripeClient->changePlan($this->getUser(), $plan);
        } catch (\Stripe\Error\Card $e) {
... lines 152 - 154
        }
... lines 156 - 161
    }
}
```

If this happens, return a new JsonResponse() with a message key set to $e->getMessage():

164 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 141
    public function changePlanAction($planId)
    {
... lines 144 - 148
        try {
            $stripeSubscription = $stripeClient->changePlan($this->getUser(), $plan);
        } catch (\Stripe\Error\Card $e) {
            return new JsonResponse([
                'message' => $e->getMessage()
            ], 400);
        }
... lines 156 - 161
    }
}
```

This will be something like: "Your card was declined".

Oh, and give this a 400 status code so that jQuery knows that this AJAX call has failed.

Finally, in the template, add a .fail() callback with a jqXHR argument:

158 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script>
        jQuery(document).ready(function() {
... lines 10 - 15
            $('.js-change-plan-button').on('click', function(e) {
... lines 17 - 24
                $.ajax({
                    url: previewUrl
                }).done(function(data) {
... lines 28 - 34
                    swal({
... lines 36 - 41
                    }, function () {
                        $.ajax({
                            url: changeUrl,
                            method: 'POST'
                        }).done(function() {
... lines 47 - 52
                        }).fail(function(jqXHR) {
                            swal({
                                title: 'Plan change failed!',
                                text: jqXHR.responseJSON.message,
                                type: 'error'
                            });
                        });
                        // todo - actually change the plan!
                    });
                });
            })
        });
    </script>
{% endblock %}
... lines 67 - 158
```

I'll paste in one last sweet alert popup that shows the message to the user.

Give it a Floor Run, See if it Plays

Let's test this *whole* big mess. Our current subscription is totally messed up, so go add a new, fresh Farmer Brent to your cart. Then, checkout with the functional, fake card.

Cool! In the account page, update the card to the one that will fail.

Before we upgrade, refresh the Customer page in Stripe to see how it looks. First, there's no customer balance, and our current subscription is for the Farmer Brent.

Ok, upgrade to the New Zealander! And... Plan change failed! That looks bad, but it's wonderful!

Reload the Customer page. First, the customer still has no account balance, that's good. Second, we can see the failed payment, but we're still on the Farmer Brent plan. And the $100 invoice is unpaid, but it's *closed*. Stripe won't try to pay this again.

Back on the Customer page, find Events at the bottom and click to view more. This tells the *whole* story: we upgraded to the New Zealander plan, the two proration invoice items were created, the invoice was created, the invoice payment failed, we updated the invoice to be closed, and finally downgraded back to the Farmer Brent plan.

WOW. Go find a co-worker and challenge them to break your setup. We are now, truly, rock solid.

Chapter 28: Changing your Plan from Monthly to Yearly

So far, we *only* offer monthly plans. But sheep *love* commitment, so they've been asking for *yearly* options. Well, great news! After all that upgrade stuff we just handled, this is going to be *easy*.

Creating the New Plans

First, in Stripe's dashboard, we need to create two new plans. Call the first "Farmer Brent yearly" and for the total... how about 99 X 10: so $990, per year.

Then, add the New Zealander yearly, set to 1990, billed yearly.

Cool! I'm not going to update our checkout to allow these plans initially, because, honestly, that's super easy: just create some new links to add these plans to your cart, and you're done.

Nope, we'll skip straight to the hard stuff: allowing the user to *change* between monthly and yearly plans.

Adding the SubscriptionPlan Objects

First, we need to add these plans to our system. Open the SubscriptionPlan class. To distinguish between monthly and yearly plans, add a new property called duration: this will be a string, either monthly or yearly. At the top, I love constants, so create: const DURATION_MONTHLY = 'monthly' and const DURATION_YEARLY = 'yearly':

45 lines | src/AppBundle/Subscription/SubscriptionPlan.php

```
... lines 1 - 4
class SubscriptionPlan
{
    const DURATION_MONTHLY = 'monthly';
    const DURATION_YEARLY = 'yearly';
... lines 9 - 44
}
```

Next, add a $duration argument to the constructor, but default it to monthly. Set the property below:

45 lines | src/AppBundle/Subscription/SubscriptionPlan.php

```
... lines 1 - 4
class SubscriptionPlan
{
... lines 7 - 15
    private $duration;

    public function __construct($planId, $name, $price, $duration = self::DURATION_MONTHLY)
    {
... lines 20 - 22
        $this->duration = $duration;
    }
... lines 25 - 44
}
```

Finally, I'll use the "Code"->"Generate" menu, or Command+N on a Mac, select "Getters" and then choose duration. That gives me a nice getDuration() method:

45 lines | src/AppBundle/Subscription/SubscriptionPlan.php

```
... lines 1 - 4
class SubscriptionPlan
{
... lines 7 - 40
    public function getDuration()
    {
        return $this->duration;
    }
}
```

In SubscriptionHelper, we create and preload all of our plans. Copy the two monthly plans, paste them, update their keys to have yearly and add the last argument for the *yearly* duration:

133 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 15
    public function __construct(EntityManager $em)
    {
... lines 18 - 31
        $this->plans[] = new SubscriptionPlan(
            'farmer_brent_yearly',
            'Farmer Brent',
            990,
            SubscriptionPlan::DURATION_YEARLY
        );

        $this->plans[] = new SubscriptionPlan(
            'new_zealander_yearly',
            'New Zealander',
            1990,
            SubscriptionPlan::DURATION_YEARLY
        );
    }
... lines 46 - 131
}
```

Now, these are *at least* valid plans inside the system.

The Duration Change UI

Here's the goal: on the account page, next to the "Next Billing at" text, I want to add a link that says "bill yearly" or "bill monthly". When you click this, it should follow the same workflow we just built for *upgrading* a plan: it should show the cost, then make the change.

In ProfileController::accountAction(), add yet *another* variable here called $otherDurationPlan:

170 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
15  class ProfileController extends BaseController
16  {
... lines 17 - 19
20      public function accountAction()
21      {
22          $currentPlan = null;
23          $otherPlan = null;
24          $otherDurationPlan = null;
... lines 25 - 42
43      }
... lines 44 - 168
169 }
```

This will eventually be the SubscriptionPlan object for the *other* duration of the current plan. So if I have the *monthly* Farmer Brent, this will be set to the *yearly* Farmer Brent plan.

To find that plan, open SubscriptionHelper and add a new function called findPlanForOtherDuration() with a $currentPlanId argument:

133 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
9   class SubscriptionHelper
10  {
... lines 11 - 74
75      public function findPlanForOtherDuration($currentPlanId)
76      {
77          if (strpos($currentPlanId, 'monthly') !== false) {
78              $newPlanId = str_replace('monthly', 'yearly', $currentPlanId);
79          } else {
80              $newPlanId = str_replace('yearly', 'monthly', $currentPlanId);
81          }
82
83          return $this->findPlan($newPlanId);
84      }
... lines 85 - 131
132 }
```

I'll paste in some silly code here. This relies on our naming conventions to switch between monthly and yearly plans.

Back in the controller, copy the $otherPlan line, paste it, then update the variable to $otherDurationPlan and the method to findPlanForOtherDuration():

170 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
15  class ProfileController extends BaseController
16  {
... lines 17 - 19
20      public function accountAction()
21      {
... lines 22 - 23
24          $otherDurationPlan = null;
25          if ($this->getUser()->hasActiveSubscription()) {
... lines 26 - 31
32              $otherDurationPlan = $this->get('subscription_helper')
33                  ->findPlanForOtherDuration($currentPlan->getPlanId());
34          }
... lines 35 - 42
43      }
... lines 44 - 168
169 }
```

Pass that into the template as a new variable:

170 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
15  class ProfileController extends BaseController
16  {
... lines 17 - 19
20      public function accountAction()
21      {
... lines 22 - 23
24          $otherDurationPlan = null;
25          if ($this->getUser()->hasActiveSubscription()) {
... lines 26 - 31
32              $otherDurationPlan = $this->get('subscription_helper')
33                  ->findPlanForOtherDuration($currentPlan->getPlanId());
34          }
35
36          return $this->render('profile/account.html.twig', [
... lines 37 - 40
41              'otherDurationPlan' => $otherDurationPlan,
42          ]);
43      }
... lines 44 - 168
169 }
```

Cool!

In account.html.twig, scroll down to the Upgrade Plan button. Copy that *whole* thing. Then, keep scrolling to the "Next Billing at" section. If the user has a subscription, paste the upgrade button:

166 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 67
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 73 - 88
                <table class="table">
                    <tbody>
... lines 91 - 116
                        <tr>
                            <th>Next Billing at:</th>
                            <td>
                                {% if app.user.hasActiveNonCancelledSubscription %}
                                    {{ app.user.subscription.billingPeriodEndsAt|date('F jS') }}

                                    <button class="btn btn-xs btn-link pull-right js-change-plan-button"
                                        data-preview-url="{{ path('account_preview_plan_change', {'planId': otherDurationPlan.planId}) }}"
                                        data-plan-name="{{ otherDurationPlan.name }} {{ otherDurationPlan.duration }}"
                                        data-change-url="{{ path('account_execute_plan_change', {'planId': otherDurationPlan.planId}) }}"
                                    >
                                        Bill {{ otherDurationPlan.duration }}
                                    </button>
... lines 130 - 131
                                {% endif %}
                            </td>
                        </tr>
... lines 135 - 148
                    </tbody>
                </table>
            </div>
... lines 152 - 160
        </div>
    </div>
</div>
{% endblock %}
... lines 165 - 166
```

And since *this* process will be the same as upgrading, we can re-use this exactly. Just change otherPlan to otherDurationPlan... in all 4 places. Update the text to "Bill" and then otherDurationPlan.duration. So, this will say something like "Bill yearly".

Dump the Upcoming Invoice

Before we try this, go back into ProfileController and find previewPlanChangeAction(). The truth is, changing a plan from monthly to yearly should be *identical* to upgrading a plan. But, it's not *quite* the same. To help us debug an issue we're about to see, dump the $stripeInvoice variable:

170 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 122
    public function previewPlanChangeAction($planId)
    {
... lines 125 - 127
        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );
        dump($stripeInvoice);
... lines 134 - 141
    }
... lines 143 - 168
}
```

And now that I've told you it won't work, let's try it out! Refresh the account page. Then click the new "Bill yearly" link. Ok:

> You will be charged $792.05 immediately

Wait, that doesn't seem right. The yearly plan is $990 per year. Then, if you subtract approximately $99 from that as a credit, it should be something closer to $891. Something is not quite right.

Chapter 29: Monthly to Yearly: The Billing Period Change

We just found out that this amount - $792 - doesn't seem right! Open the web debug toolbar and click to see the profiler for the "preview change" AJAX call that returned this number. Click the Debug link on the left. This is a dump of the upcoming invoice. And according to *it*, the user will owe $891.05. Wait, that sounds *exactly* right!

The number we just saw is different because, remember, we start with amount_due, but then subtract the plan total to remove the extra line item that Stripe adds. Back then, we had *three* line items: the two prorations and a line item for the next, full month.

But woh, now there's only two line items: the partial-month discount and a charge for the *full*, yearly period.

Changing Duration changes Billing Period

Stripe talks about this oddity in their documentation: when you change to a plan with a different duration - so monthly to yearly or vice-versa - Stripe bills you *immediately* and changes your billing date to start *today*.

So if you're normally billed on the first of the month and you change from monthly to yearly on the 15th, you'll be credited half of your monthly subscription and then charged for a full year. That yearly subscription will start immediately, on the 15th of that month and be renewed in one year, on the 15th.

For us, this means that the amount_due on the Invoice is actually correct: we don't need to adjust it. In ProfileController, create a new variable called $currentUserPlan set to $this->get('subscription_helper')->findPlan() and pass it $this->getUser()->getSubscription()->getStripePlanId():

175 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 122
    public function previewPlanChangeAction($planId)
    {
... lines 125 - 127
        $stripeInvoice = $this->get('stripe_client')
            ->getUpcomingInvoiceForChangedSubscription(
                $this->getUser(),
                $plan
            );

        $currentUserPlan = $this->get('subscription_helper')
            ->findPlan($this->getUser()->getSubscription()->getStripePlanId());
... lines 136 - 146
    }
... lines 148 - 173
}
```

Now, if $plan - which is the new plan - $plan->getDuration() matches the $currentUserPlan->getDuration(), then we *should* correct the total. Otherwise, if the duration is changing, the $total is already perfect:

175 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 122
    public function previewPlanChangeAction($planId)
    {
... lines 125 - 133
        $currentUserPlan = $this->get('subscription_helper')
            ->findPlan($this->getUser()->getSubscription()->getStripePlanId());

        $total = $stripeInvoice->amount_due;

        // contains the pro-rations
        // *plus* - if the duration matches - next cycle's amount
        if ($plan->getDuration() == $currentUserPlan->getDuration()) {
            // subtract plan price to *remove* next the next cycle's total
            $total -= $plan->getPrice() * 100;
        }
... lines 145 - 146
    }
... lines 148 - 173
}
```

Since this looks *totally* weird, I'll tweak my comment to mention that amount_due contains the extra month charge *only* if the duration stays the same.

Ok! Go back and refresh! Click "Bill yearly". Yes! That looks right: $891.06.

Duration Change? Don't Invoice

Because of this behavior difference when the duration changes, we need to fix *one* other spot: in StripeClient::changePlan(). Right now, we manually create an invoice so that the customer is charged immediately. But... we don't need to do that in this case: Stripe *automatically* creates and pays an Invoice when the duration changes.

In fact, trying to create an invoice will throw an error! Let's see it. First, update your credit card to one that will work.

Now, change to Bill yearly and confirm. The AJAX call *should* fail... and it does! Open the profiler for that request and find the Exception:

> Nothing to invoice for customer

Obviously, we need to avoid this. In StripeClient, add a new variable: $currentPeriodStart that's set to $stripeSubscription->current_period_start:

184 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
        $stripeSubscription = $this->findSubscription($user->getSubscription()->getStripeSubscriptionId());

        $currentPeriodStart = $stripeSubscription->current_period_start;
... lines 159 - 181
    }
}
```

That's the current period start date *before* we change the plan.

After we change the plan, if the duration is different, the current period start will have changed. Surround the *entire* invoicing block with if $stripeSubscription->current_period_start == $currentPeriodStart:

184 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 156 - 157
        $currentPeriodStart = $stripeSubscription->current_period_start;
... lines 159 - 166
        if ($stripeSubscription->current_period_start == $currentPeriodStart) {
            try {
                // immediately invoice them
                $this->createInvoice($user);
            } catch (\Stripe\Error\Card $e) {
                $stripeSubscription->plan = $originalPlanId;
                // prevent prorations discounts/charges from changing back
                $stripeSubscription->prorate = false;
                $stripeSubscription->save();

                throw $e;
            }
        }
... lines 180 - 181
    }
}
```

In other words: only invoice the customer manually if the subscription period hasn't changed. I think we should add a note above this: this can look really confusing!

184 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 153
    public function changePlan(User $user, SubscriptionPlan $newPlan)
    {
... lines 156 - 157
        $currentPeriodStart = $stripeSubscription->current_period_start;
... lines 159 - 163
        // if the duration did not change, Stripe will not charge them immediately
        // but we *do* want them to be charged immediately
        // if the duration changed, an invoice was already created and paid
        if ($stripeSubscription->current_period_start == $currentPeriodStart) {
            try {
                // immediately invoice them
                $this->createInvoice($user);
            } catch (\Stripe\Error\Card $e) {
                $stripeSubscription->plan = $originalPlanId;
                // prevent prorations discounts/charges from changing back
                $stripeSubscription->prorate = false;
                $stripeSubscription->save();

                throw $e;
            }
        }
... lines 180 - 181
    }
}
```

Take it for a Test Drive

But, now it should work! Reset things by going to the pricing page and buying a brand new monthly subscription. Now, head to your account page and update it to yearly. The amount - $891 - looks right, so hit OK.

Yes! Plan changed! My option changed to "Bill monthly" and the "Next Billing at" date is August 10th - one year from today. We should *probably* print the year.

In Stripe, under payments, we have one for $891, and the customer is on the Farmer Brent *yearly* plan.

Winning!

Chapter 30: Coupons! Adding the Form

Let's talk about something fun: coupon codes. When a user checks out, I want them to be able to add a coupon code. Fortunately Stripe *totally* supports this, and that makes our job easier.

In fact, in the Stripe dashboard, there's a Coupon section. Create a new coupon: you can choose either a percent off or an amount off. To make things simple, I'm *only* going to support coupons that are for a specific *amount* off.

Create one that saves us $50. Oh, and in case this is used on an order with a subscription, you can set the duration to once, multi-month or forever.

Then, give the code a creative, unique code: like CHEAP_SHEEP. Oh, and the coupon codes *are* case sensitive.

Tip

I'm choosing to create my coupons through Stripe's admin interface. If you want an admin section that does this on *your* site, that's totally possible! You can create new Coupons through Stripe's API.

Adding a Coupon During Checkout

Back on our site, before we get into any Stripe integration, we need to add a spot for adding coupons during checkout.

Open the template: order/checkout.html.twig. Below the cart table, add a button, give it some styling classes and a js-show-code-form class. Say, "I have a coupon code":

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 26
            <div class="col-xs-12 col-sm-6">
... lines 28 - 57
                <button class="btn btn-xs btn-link pull-right js-show-code-form">
                    I have a coupon code
                </button>
... lines 61 - 75
            </div>
... lines 77 - 79
        </div>
    </div>
</div>
{% endblock %}
```

Instead of adding this form by hand, open your tutorial/ directory: this is included in the code download. Open coupon-form.twig, copy its code, then paste it below the button:

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 26
            <div class="col-xs-12 col-sm-6">
... lines 28 - 57
                <button class="btn btn-xs btn-link pull-right js-show-code-form">
                    I have a coupon code
                </button>

                <div class="js-code-form" style="display: none;">
                    <form action="" method="POST" class="form-inline">
                        <div class="form-group">
                            <div class="input-group">
                                <span class="input-group-addon">
                                    <i class="fa fa-terminal"></i>
                                </span>
                                <input type="text" name="code" autocomplete="off" class="form-control" placeholder="Coupon Code"/>
                            </div>

                            <button type="submit" class="btn btn-primary">Add</button>
                        </div>
                    </form>
                </div>
            </div>
... lines 77 - 79
        </div>
    </div>
</div>
{% endblock %}
```

This new div is hidden by default and has a js-code-form class that we'll use soon via JavaScript. And, it has just one field named code.

Copy the js-show-code-form class and scroll up to the javascripts block. Add a new document.ready() function:

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

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script>
        jQuery(document).ready(function() {
... lines 11 - 15
        });
    </script>
{% endblock %}
... lines 19 - 84
```

Inside, find the .js-show-code-form element and on click, add a callback. Start with our favorite e.preventDefault():

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

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script>
        jQuery(document).ready(function() {
            $('.js-show-code-form').on('click', function(e) {
                e.preventDefault();
... lines 13 - 14
            })
        });
    </script>
{% endblock %}
... lines 19 - 84
```

Then, scroll down to the form, copy the js-code-form class, use jQuery to select this, and... drumroll... show it!

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

```
... lines 1 - 3
{% block javascripts %}
... lines 5 - 8
    <script>
        jQuery(document).ready(function() {
            $('.js-show-code-form').on('click', function(e) {
                e.preventDefault();

                $('.js-code-form').show();
            })
        });
    </script>
{% endblock %}
... lines 19 - 84
```

Cool! Now when you refresh, we have a new link that shows the form.

Submitting the Coupon Form

So let's move to phase two: when we hit "Add", this should submit to a new endpoint that validates the code in Stripe and attaches it to our user's cart.

To create the new endpoint, open OrderController. Near the bottom add a new public function addCouponAction() with @Route("/checkout/coupon"). Name it order_add_coupon. And to be extra-hipster, add @Method("POST") to guarantee that you can only POST to this:

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

```
... lines 1 - 9
use Symfony\Component\HttpFoundation\Request;

class OrderController extends BaseController
{
... lines 14 - 79
    /**
     * @Route("/checkout/coupon", name="order_add_coupon")
     * @Method("POST")
     */
    public function addCouponAction(Request $request)
    {
... lines 86 - 97
    }
... lines 99 - 144
}
... lines 146 - 147
```

Cool! Copy the route name, then find the coupon form in the checkout template. Update the form's action: add path() and then paste the route name:

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 26
            <div class="col-xs-12 col-sm-6">
... lines 28 - 61
                <div class="js-code-form" style="display: none;">
                    <form action="{{ path('order_add_coupon') }}" method="POST" class="form-inline">
... lines 64 - 73
                    </form>
                </div>
            </div>
... lines 77 - 79
        </div>
    </div>
</div>
{% endblock %}
```

Next, we'll read the submitted code and check with Stripe to make sure it's real, and not just someone trying to guess clever coupon codes. Come on, we've all tried it before.

Chapter 31: Validate that Coupon in Stripe!

The coupon form will submit its code field right here:

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

```
... lines 1 - 9
10  use Symfony\Component\HttpFoundation\Request;
11
12  class OrderController extends BaseController
13  {
... lines 14 - 83
84      public function addCouponAction(Request $request)
85      {
... lines 86 - 97
98      }
... lines 99 - 144
145 }
... lines 146 - 147
```

To fetch that POST parameter, add the Request object as an argument. Then add $code = $request->request->get('code'):

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

```
... lines 1 - 9
10  use Symfony\Component\HttpFoundation\Request;
11
12  class OrderController extends BaseController
13  {
... lines 14 - 83
84      public function addCouponAction(Request $request)
85      {
86          $code = $request->request->get('code');
... lines 87 - 97
98      }
... lines 99 - 144
145 }
... lines 146 - 147
```

And in case some curious user submits while the form is empty, send back a validation message with $this->addFlash('error', 'Missing coupon code'). Redirect to the checkout page:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    public function addCouponAction(Request $request)
    {
        $code = $request->request->get('code');

        if (!$code) {
            $this->addFlash('error', 'Missing coupon code!');

            return $this->redirectToRoute('order_checkout');
        }
... lines 93 - 97
    }
... lines 99 - 144
}
... lines 146 - 147
```

Fetching the Coupon Information

Great! At this point, all *we* need to do is talk to Stripe and ask them:

> Hey Stripe! Is this a valid coupon code in your system?. Oh, and if it is, ow much is it for?

Since Coupon is just an object in Stripe's API, we can fetch it like *anything* else. Booya!

As usual, add the API call in StripeClient. At the bottom, create a new public function called findCoupon() with a $code argument:

193 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 183
    /**
     * @param $code
     * @return \Stripe\Coupon
     */
    public function findCoupon($code)
    {
... line 190
    }
}
```

Then, return \Stripe\Coupon::retrieve() and pass it the $code string, which is the Coupon's primary key in Stripe's API:

193 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 187
    public function findCoupon($code)
    {
        return \Stripe\Coupon::retrieve($code);
    }
}
```

Back in OrderController, add $stripeCoupon = $this->get('stripe_client') and then call ->findCoupon($code):

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    public function addCouponAction(Request $request)
    {
        $code = $request->request->get('code');

        if (!$code) {
            $this->addFlash('error', 'Missing coupon code!');

            return $this->redirectToRoute('order_checkout');
        }

        $stripeCoupon = $this->get('stripe_client')
            ->findCoupon($code);

        dump($stripeCoupon);die;
    }
... lines 99 - 144
}
... lines 146 - 147
```

If the code is invalid, Stripe will throw an exception. We'll handle that in a few minutes. But just for now, let's dump($stripeCoupon) and die to see what it looks like.

Ok, refresh, hit "I have a coupon code," fill in our CHEAP_SHEEP code, and submit!

There it is! In the _values section where the data hides, the coupon has an id, it shows the amount_off in cents and has a few other things, like duration, in case you want to create coupons that are recurring and need to tell the user that this will be applied multiple times.

Now that we know the coupon is legit, we should add it to our cart. I've already prepped the cart to be able to store coupons. Just use $this->get('shopping_cart') and then call ->setCouponCode(), passing it the $code string and the amount off, in dollars: so $stripeCoupon->amount_off/100:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    public function addCouponAction(Request $request)
    {
... lines 86 - 93
        $stripeCoupon = $this->get('stripe_client')
            ->findCoupon($code);

        $this->get('shopping_cart')
            ->setCouponCode($code, $stripeCoupon->amount_off / 100);
... lines 99 - 102
    }
... lines 104 - 149
}
... lines 151 - 152
```

The cart will *remember* - via the session - that the user has this coupon.

We're *just* about done: add a *sweet* flash message - "Coupon applied!" - and then redirect back to the checkout page:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    public function addCouponAction(Request $request)
    {
... lines 86 - 96
        $this->get('shopping_cart')
            ->setCouponCode($code, $stripeCoupon->amount_off / 100);

        $this->addFlash('success', 'Coupon applied!');

        return $this->redirectToRoute('order_checkout');
    }
... lines 104 - 149
}
... lines 151 - 152
```

Showing the Code on Checkout

Refresh and re-POST the form! Coupon applied! Except... I don't see *any* difference: the total is *still* $99.

Here's why: it's specific to our ShoppingCart object. In checkout.html.twig, we print cart.total:

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 26
            <div class="col-xs-12 col-sm-6">
                <table class="table table-bordered">
... lines 29 - 49
                    <tfoot>
                        <tr>
                            <th class="col-xs-6 info">Total</th>
                            <td class="col-xs-3 info checkout-total">${{ cart.total }}</td>
                        </tr>
                    </tfoot>
                </table>
... lines 57 - 75
            </div>
... lines 77 - 79
        </div>
    </div>
</div>
{% endblock %}
```

I designed the ShoppingCart class so that the getTotal() method adds up *all* of the product prices plus the subscription total:

155 lines | src/AppBundle/Store/ShoppingCart.php

```
... lines 1 - 9
class ShoppingCart
{
... lines 12 - 83
    public function getTotal()
    {
        $total = 0;
        foreach ($this->getProducts() as $product) {
            $total += $product->getPrice();
        }

        if ($this->getSubscriptionPlan()) {
            $price = $this->getSubscriptionPlan()
                ->getPrice();

            $total += $price;
        }

        return $total;
    }
... lines 100 - 153
}
```

But, *this* method doesn't subtract the coupon discount. I did this to keep things clean: total is really more like a "sub-total".

But no worries, the method below this - getTotalWithDiscount() - subtracts the coupon code:

155 lines | src/AppBundle/Store/ShoppingCart.php

```
... lines 1 - 9
class ShoppingCart
{
... lines 12 - 100
    public function getTotalWithDiscount()
    {
        return max($this->getTotal() - $this->getCouponCodeValue(), 0);
    }
... lines 105 - 153
}
```

So back in the template, use cart.totalWithDiscount:

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 26
            <div class="col-xs-12 col-sm-6">
                <table class="table table-bordered">
... lines 29 - 56
                    <tfoot>
                        <tr>
                            <th class="col-xs-6 info">Total</th>
                            <td class="col-xs-3 info checkout-total">${{ cart.totalWithDiscount }}</td>
                        </tr>
                    </tfoot>
                </table>
... lines 64 - 82
            </div>
... lines 84 - 86
        </div>
    </div>
</div>
{% endblock %}
```

Ah, *now* it shows $49.

But, it'll be even *clearer* if we display the discount in the table. At the bottom of that table, add a new if statement: if cart.couponCode and an endif. Then, copy the subscription block from above, paste it here, and change the first variable to cart.couponCode and the second to cart.couponCodeValue without the / month, unless you want to make all your coupons recurring. Oh, and add "Coupon" in front of the code:

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 26
            <div class="col-xs-12 col-sm-6">
                <table class="table table-bordered">
... lines 29 - 34
                    <tbody>
... lines 36 - 49
                        {% if cart.couponCode %}
                            <tr>
                                <th class="col-xs-6 checkout-product-name">Coupon {{ cart.couponCode }}</th>
                                <td class="col-xs-3">${{ cart.couponCodeValue }}</td>
                            </tr>
                        {% endif %}
                    </tbody>
                    <tfoot>
                        <tr>
                            <th class="col-xs-6 info">Total</th>
                            <td class="col-xs-3 info checkout-total">${{ cart.totalWithDiscount }}</td>
                        </tr>
                    </tfoot>
                </table>
... lines 64 - 82
            </div>
... lines 84 - 86
        </div>
    </div>
</div>
{% endblock %}
```

This time, the whole page makes sense! $99 - $50 = $49. It's a miracle!

Now for the easy step: apply the coupon to the user's order at checkout... ya know, so that they *actually* save $50.

Chapter 32: Applying a Coupon at Checkout

There are two ways to use a coupon on checkout: either attach it to the *subscription* to say "This subscription should have this coupon code" - *or* - attach it to the *customer*. They're approximately the same, but we'll attach the coupon to the customer, in part, because the coupon should *also* work on individual products.

In OrderController, scroll down to the chargeCustomer() method:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 108
    private function chargeCustomer($token)
    {
        $stripeClient = $this->get('stripe_client');
        /** @var User $user */
        $user = $this->getUser();
        if (!$user->getStripeCustomerId()) {
            $stripeCustomer = $stripeClient->createCustomer($user, $token);
        } else {
            $stripeCustomer = $stripeClient->updateCustomerCard($user, $token);
        }

        // save card details
        $this->get('subscription_helper')
            ->updateCardDetails($user, $stripeCustomer);

        $cart = $this->get('shopping_cart');

        foreach ($cart->getProducts() as $product) {
            $stripeClient->createInvoiceItem(
                $product->getPrice() * 100,
                $user,
                $product->getName()
            );
        }

        if ($cart->getSubscriptionPlan()) {
            // a subscription creates an invoice
            $stripeSubscription = $stripeClient->createSubscription(
                $user,
                $cart->getSubscriptionPlan()
            );

            $this->get('subscription_helper')->addSubscriptionToUser(
                $stripeSubscription,
                $user
            );
        } else {
            // charge the invoice!
            $stripeClient->createInvoice($user, true);
        }
    }
}
... lines 151 - 152
```

We know this method: we get or create the Stripe Customer, create InvoiceItems for any products, create the Subscription, and then create an invoice, if needed.

Before adding the invoice items, let's add the coupon to the Customer. So, if $cart->getCouponCodeValue(), then very simply, $stripeCustomer->coupon = $cart->getCouponCode(). Make it official with $customer->save():

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 108
109     private function chargeCustomer($token)
110     {
... lines 111 - 123
124         $cart = $this->get('shopping_cart');
125
126         if ($cart->getCouponCodeValue()) {
127             $stripeCustomer->coupon = $cart->getCouponCode();
128             $stripeCustomer->save();
129         }
... lines 130 - 153
154     }
155 }
... lines 156 - 157
```

The important thing is that *you* don't need to change how much you're charging the user: attach the coupon, charge them for the full amount, and let Stripe figure it all out.

I think we should try this out! Use our favorite fake credit card, and Checkout! So far so good!

Find the Customer in Stripe. Yep! There's the order: $49. The invoice tells the *whole* story: with the sub-total, the discount and the total.

Very, very, nice.

Handling Invalid Coupons

And very easy! So easy, that we have time to add code to handle *invalid* coupons. Add another item to your cart. Now, try a FAKE coupon code.

Ah! 500 error is *no* fun. The exception is a \Stripe\Error\InvalidRequest because, basically, the API responds with a 404 status code.

This all falls apart in OrderController on line 95. Hunt that down!

Ah, findCoupon(): surround this beast with a try-catch block for \Stripe\Error\InvalidRequest:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 83
84      public function addCouponAction(Request $request)
85      {
... lines 86 - 93
94          try {
95              $stripeCoupon = $this->get('stripe_client')
96                  ->findCoupon($code);
97          } catch (\Stripe\Error\InvalidRequest $e) {
... lines 98 - 100
101         }
... lines 102 - 108
109     }
... lines 110 - 160
161 }
... lines 162 - 163
```

The easiest thing to do is add a flash error message: Invalid Coupon code. Then, redirect back to the checkout page:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 83
84      public function addCouponAction(Request $request)
85      {
... lines 86 - 93
94          try {
95              $stripeCoupon = $this->get('stripe_client')
96                  ->findCoupon($code);
97          } catch (\Stripe\Error\InvalidRequest $e) {
98              $this->addFlash('error', 'Invalid coupon code!');
99
100             return $this->redirectToRoute('order_checkout');
101         }
... lines 102 - 108
109     }
... lines 110 - 160
161 }
... lines 162 - 163
```

Refresh that bad coupon! Ok! That's covered!

Expired Coupons

There's just *one* other situation to handle. In Stripe, find the Coupon section and create a second code. Set the amount to $50, duration "once" and the code: SINGLE_USE. By here's the kicker: set Max redemptions to 1. So, only *one* customer should be able to use this. There's also a time-sensitive "Redeem by" option.

Quickly, go use the SINGLE_USE code and fill out the form to checkout. This will be the first - and only - allowed "redemption" of this code. When you refresh the Coupon page in Stripe, Redemptions are 1/1.

Now, add another subscription to your cart. If you tried to use the code a *second* time, our system *would* allow this. And that makes sense: *all* we're doing now is looking up the code in Stripe to make sure it *exists*.

But, if we tried to checkout, Stripe would be *pissed*: it would *not* allow us to use the code a second time. Stripe has our back.

But, we should *definitely* prevent the code from being attached to the cart in the first place. Checkout the Coupon section of Stripe's API docs. Ah, this valid field is the *key*. This field basically answers this question:

> In this moment, can this coupon be used?

Brilliant! Back in OrderController::addCouponAction(), add an if statement: if !$stripeCoupon->valid, then, just like in the catch, add an error flash - "Coupon expired" - and redirect over to the checkout page:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 83
    public function addCouponAction(Request $request)
    {
... lines 86 - 93
        try {
            $stripeCoupon = $this->get('stripe_client')
                ->findCoupon($code);
        } catch (\Stripe\Error\InvalidRequest $e) {
            $this->addFlash('error', 'Invalid coupon code!');

            return $this->redirectToRoute('order_checkout');
        }

        if (!$stripeCoupon->valid) {
            $this->addFlash('error', 'Coupon expired');

            return $this->redirectToRoute('order_checkout');
        }
... lines 108 - 114
    }
... lines 116 - 166
}
... lines 168 - 169
```

Try it again. Awesome, this time, we get blocked.

If you want to be *extra* careful, you could add some try-catch logic to your checkout code *just* to prevent the edge-case where the code becomes *invalid* between the time of adding it to your cart and checking out. But either way, Stripe will *never* allow an invalid coupon to be used.

Chapter 33: Free (Ice Cream) Checkout!

Go create a *huge* coupon - like for $500. Call this one SUPER_SHEEP!

Ok, this coupon is awesome - so let's add it to an order.

Perfect! As you can see, the cart is *already* smart enough to return the total as $0, instead of a negative number. Way to go cart!

But, uh, what's this checkout form still doing over here? Why should I need to enter my credit card info? This order is *free*!

Look, how you handle *free* orders is up to you. If you *still* want to require a credit card, you can do that. The customer won't be charged, but the card will be on file for renewals.

But that's not for me: I want to make it as easy as possible for this user to checkout free.

Hide the Checkout Form

The first step, is to hide this checkout form! In checkout.html.twig, find the _cardForm.html.twig include, and wrap it in if cart.totalWithDiscount > 0:

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 83
            <div class="col-xs-12 col-sm-6">
                {% if cart.totalWithDiscount > 0 %}
                    {{ include('order/_cardForm.html.twig') }}
                {% else %}
... lines 88 - 92
                {% endif %}
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Else, create a really simple form that submits right back to this URL. Give it method="POST" and a submit button that invites the user to "Checkout for Free":

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

```
... lines 1 - 19
{% block body %}
<div class="nav-space-checkout">
    <div class="container">
        <div class="row">
... lines 24 - 83
            <div class="col-xs-12 col-sm-6">
                {% if cart.totalWithDiscount > 0 %}
                    {{ include('order/_cardForm.html.twig') }}
                {% else %}
                    <form action="" method="POST">
                        <button type="submit" class="btn btn-lg btn-danger">
                            Checkout for Free!
                        </button>
                    </form>
                {% endif %}
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

If you refresh now, boom! Checkout form gone.

Handling Free Checkout

But that's not *quite* everything we need to do. Thanks to the discount, Stripe will already know that it doesn't need to charge the user, and so, the customer doesn't need to have a card. But, the funny thing is, up until now, our OrderController logic is expecting a stripeToken to *always* be submitted:

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

```
... lines 1 - 11
class OrderController extends BaseController
{
... lines 14 - 48
    public function checkoutAction(Request $request)
    {
... lines 51 - 53
        if ($request->isMethod('POST')) {
            $token = $request->request->get('stripeToken');
... lines 56 - 68
        }
... lines 70 - 77
    }
... lines 79 - 166
}
... lines 168 - 169
```

Remember that's the token that we get back from Stripe, after submitting the credit card information to them. We then pass that to chargeCustomer() and attach it to the Customer:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 120
121     private function chargeCustomer($token)
122     {
... lines 123 - 125
126         if (!$user->getStripeCustomerId()) {
127             $stripeCustomer = $stripeClient->createCustomer($user, $token);
128         } else {
129             $stripeCustomer = $stripeClient->updateCustomerCard($user, $token);
130         }
... lines 131 - 165
166     }
167 }
... lines 168 - 169
```

Optionally Apply the Stripe Token

But now, our code needs to be smart enough to *not* try to attach the token to the Customer for free orders. At the top, add a *sanity* check: if there is *no* token, and the shopping cart's total with discount is *not* free... well, we have a problem! Throw a clear exception: the order is non-free... but we're missing the payment token!

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 120
121     private function chargeCustomer($token)
122     {
123         if (!$token && $this->get('shopping_cart')->getTotalWithDiscount() > 0) {
124             throw new \Exception('Somehow the order is non-free, but we have no token!?');
125         }
... lines 126 - 174
175     }
176 }
... lines 177 - 178
```

Next, when we call createCustomer(), we pass in the $token. Open StripeClient and find that method:

193 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
9   class StripeClient
10  {
... lines 11 - 19
20      public function createCustomer(User $user, $paymentToken)
21      {
22          $customer = \Stripe\Customer::create([
... line 23
24              'source' => $paymentToken,
25          ]);
... lines 26 - 31
32      }
... lines 33 - 191
192 }
```

Hmm. Now, $paymentToken might be blank. But Stripe will be *really* angry if we try to attach an empty *source* to the Customer. Instead of doing this all at once, add a new $data array variable, and move the email key into it. Then, pass $data to the create() call:

203 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 19
    public function createCustomer(User $user, $paymentToken)
    {
        $data = [
            'email' => $user->getEmail(),
        ];
... lines 25 - 29
        $customer = \Stripe\Customer::create($data);
... lines 31 - 36
    }
... lines 38 - 201
}
```

You know what's next: if $paymentToken is not blank, add a source key to $data set to $paymentToken:

203 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 19
    public function createCustomer(User $user, $paymentToken)
    {
        $data = [
            'email' => $user->getEmail(),
        ];

        if ($paymentToken) {
            $data['source'] = $paymentToken;
        }

        $customer = \Stripe\Customer::create($data);
... lines 31 - 36
    }
... lines 38 - 201
}
```

We're done here.

But we have the same problem in updateCustomerCard(). Let's fix this here: if $stripeToken, then update the customer's card:

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 120
121     private function chargeCustomer($token)
122     {
... lines 123 - 129
130         if (!$user->getStripeCustomerId()) {
131             $stripeCustomer = $stripeClient->createCustomer($user, $token);
132         } else {
133             // don't need to update it if the order is fre
134             if ($token) {
135                 $stripeCustomer = $stripeClient->updateCustomerCard($user, $token);
136             } else {
... line 137
138             }
139         }
... lines 140 - 174
175     }
176 }
... lines 177 - 178
```

Else, we *do* need to fetch the \Stripe\Customer object - we use it below. In StripeClient, add a new method to do this: public function findCustomer() with a User argument:

203 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
9   class StripeClient
10  {
... lines 11 - 48
49      public function findCustomer(User $user)
50      {
... line 51
52      }
... lines 53 - 201
202 }
```

Then, return the timeless \Stripe\Customer::retrieve($user->getStripeCustomerId()):

203 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
9   class StripeClient
10  {
... lines 11 - 48
49      public function findCustomer(User $user)
50      {
51          return \Stripe\Customer::retrieve($user->getStripeCustomerId());
52      }
... lines 53 - 201
202 }
```

In the controller, call that: $stripeCustomer = $stripeClient->findCustomer($user):

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

```
... lines 1 - 11
12  class OrderController extends BaseController
13  {
... lines 14 - 120
121     private function chargeCustomer($token)
122     {
... lines 123 - 129
130         if (!$user->getStripeCustomerId()) {
131             $stripeCustomer = $stripeClient->createCustomer($user, $token);
132         } else {
133             // don't need to update it if the order is fre
134             if ($token) {
135                 $stripeCustomer = $stripeClient->updateCustomerCard($user, $token);
136             } else {
137                 $stripeCustomer = $stripeClient->findCustomer($user);
138             }
139         }
... lines 140 - 174
175     }
176 }
... lines 177 - 178
```

Ok, I'm feeling good! The last trouble spot is in updateCardDetails(). Open SubscriptionHelper:

133 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
9   class SubscriptionHelper
10  {
... lines 11 - 104
105     public function updateCardDetails(User $user, \Stripe\Customer $stripeCustomer)
106     {
107         $cardDetails = $stripeCustomer->sources->data[0];
108         $user->setCardBrand($cardDetails->brand);
109         $user->setCardLast4($cardDetails->last4);
110         $this->em->persist($user);
111         $this->em->flush($user);
112     }
... lines 113 - 131
132 }
```

Oh yea - this method looks at the sources key on the Customer to get the card brand and last four digits. In our app, every Customer has exactly *one* card, so we use the 0 key. But guess what! Not anymore: a customer *might* have zero cards.

So we just need to code defensively: add an if statement: if !$stripeCustomer->sources->data, just return: there's no card details to update:

138 lines | src/AppBundle/Subscription/SubscriptionHelper.php

```
... lines 1 - 8
class SubscriptionHelper
{
... lines 11 - 104
    public function updateCardDetails(User $user, \Stripe\Customer $stripeCustomer)
    {
        if (!$stripeCustomer->sources->data) {
            // the customer may not have a card on file
            return;
        }
... lines 111 - 116
    }
... lines 118 - 136
}
```

Ok, we're done! A free checkout and a normal checkout are *almost* the same. The only difference is that you *don't* have a Stripe token, so you can't attach that to your Customer.

Refresh the checkout page and, "Checkout for Free".

It works! In Stripe, find our Customer. There is no new payment, but there *is* an Invoice for $0 and an active subscription. The Invoice shows off the discount.

No Card? Webhook Problems

Thanks to this change, it's now possible for a Customer to *not* have *any* cards attached in Stripe. And yea know what? This creates a new problem in a *totally* unrelated part of the process: webhooks.

But, it's no big deal. Open WebhookController and find the invoice.payment_failed section. Wait! Woh! Before that - oh geez - fix my *horrible* typo: invoice.payment_succeeded:

104 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 9
class WebhookController extends BaseController
{
... lines 12 - 14
    public function stripeWebhookAction(Request $request)
    {
... lines 17 - 43
        switch ($stripeEvent->type) {
... lines 45 - 51
            case 'invoice.payment_succeeded':
... lines 53 - 78
        }
... lines 80 - 81
    }
... lines 83 - 102
}
```

This is why you must *test* your webhooks!

Anyways, back to invoice.payment_failed. Our *entire* reason for handling this webhook is so that we can send the user an email to tell them that we're having a problem charging their card. We didn't actually do the work, but that email would probably sound like this:

> Hey friend! So, we're having a problem charging your card. If you need to update it, go to your account page and add the new details there.

But what will happen if a user checks out for free, but with a subscription? After their first month, Stripe will not be able to charge them, and it will trigger *this* webhook.

In those cases, the email should *really* have some different text, like:

> Yo amigo! I hope you enjoyed your free month. If you want to continue, you can add a credit card to your account page.

So to know *which* language to use, first fetch the Stripe Customer by saying $this->get('stripe_client')->findCustomer($user):

110 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 9
class WebhookController extends BaseController
{
... lines 12 - 14
    public function stripeWebhookAction(Request $request)
    {
... lines 17 - 43
        switch ($stripeEvent->type) {
... lines 45 - 62
            case 'invoice.payment_failed':
... lines 64 - 65
                if ($stripeSubscriptionId) {
... lines 67 - 68
                    if ($stripeEvent->data->object->attempt_count == 1) {
                        $user = $subscription->getUser();
                        $stripeCustomer = $this->get('stripe_client')
                            ->findCustomer($user->getStripeCustomerId());
... lines 73 - 77
                    }
                }
... lines 80 - 84
        }
... lines 86 - 87
    }
... lines 89 - 108
}
```

Now we can create a new variable, called $hasCardOnFile. Set that to a count of $stripeCustomer->sources->data and check if it's greater than zero:

110 lines | src/AppBundle/Controller/WebhookController.php

```
... lines 1 - 9
class WebhookController extends BaseController
{
... lines 12 - 14
    public function stripeWebhookAction(Request $request)
    {
... lines 17 - 43
        switch ($stripeEvent->type) {
... lines 45 - 62
            case 'invoice.payment_failed':
... lines 64 - 65
                if ($stripeSubscriptionId) {
... lines 67 - 68
                    if ($stripeEvent->data->object->attempt_count == 1) {
                        $user = $subscription->getUser();
                        $stripeCustomer = $this->get('stripe_client')
                            ->findCustomer($user->getStripeCustomerId());

                        $hasCardOnFile = count($stripeCustomer->sources->data) > 0;

                        // todo - send the user an email about the problem
                        // use hasCardOnFile to customize this
                    }
                }
... lines 80 - 84
        }
... lines 86 - 87
    }
... lines 89 - 108
}
```

Now, you can use that variable to write the most uplifting, majestic, and encouraging payment failed emails that the world has ever seen.

Chapter 34: Sweet Invoices

Ha! We've made it! We've survived our subscription payment setup! So, umm, go celebrate: eat some cake! Sing a song! Or, like we'll do, do some accounting.

Because our *last* topic is about that: your users *will* need to see a receipt after purchase. And good news: we've setup our system so that every charge is done by creating an *Invoice* in Stripe's system. That'll make our life easy.

Store Invoices Locally?

In fact, *all* the information we need to render a receipt is *already* stored in Stripe. So, you *could* create a local invoices database table and store details there... or, you can take a shortcut and use Stripe's API to fetch invoice data whenever you actually need it. Let's do that.

Fetch all Paid Invoices

Open up StripeClient. At the bottom, add a new public function findPaidInvoices() with a User argument:

225 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 8
class StripeClient
{
... lines 11 - 206
    public function findPaidInvoices(User $user)
    {
... lines 209 - 222
    }
}
```

Here's the idea: we'll use Stripe's API to find *all* Invoices for a customer, but then filter those to only return invoices that were *paid*. That will remove some *garbage* invoices the user shouldn't see: like invoices for payments that failed and then were closed immediately.

Start with: $allInvoices = \Stripe\Invoice::all() and pass that an array with a customer key set to $user->getStripeCustomerId():

225 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 206
    public function findPaidInvoices(User $user)
    {
        $allInvoices = \Stripe\Invoice::all([
            'customer' => $user->getStripeCustomerId()
        ]);
... lines 212 - 222
    }
}
```

Next - and this will look a little weird at first - create an $iterator variable set to $allInvoices->autoPagingIterator():

225 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 206
    public function findPaidInvoices(User $user)
    {
        $allInvoices = \Stripe\Invoice::all([
            'customer' => $user->getStripeCustomerId()
        ]);

        $iterator = $allInvoices->autoPagingIterator();
... lines 214 - 222
    }
}
```

This is actually *really* cool: if the user has a *lot* of invoices, then Stripe will *paginate* your results. But with the iterator, it will automatically make new API calls behind-the-scenes, allowing us to loop over *every* invoice, no matter how many there are.

Let's do that: start with $invoices = array(). Then foreach over $iterator as $invoice:

225 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 206
    public function findPaidInvoices(User $user)
    {
        $allInvoices = \Stripe\Invoice::all([
            'customer' => $user->getStripeCustomerId()
        ]);

        $iterator = $allInvoices->autoPagingIterator();

        $invoices = [];
        foreach ($iterator as $invoice) {
... lines 217 - 219
        }
... lines 221 - 222
    }
}
```

Very simply, we want to know if this invoice is *paid*. If $invoice->paid, then add this to the $invoices array. Finally, return those paid $invoices at the bottom:

225 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 206
    public function findPaidInvoices(User $user)
    {
        $allInvoices = \Stripe\Invoice::all([
            'customer' => $user->getStripeCustomerId()
        ]);

        $iterator = $allInvoices->autoPagingIterator();

        $invoices = [];
        foreach ($iterator as $invoice) {
            if ($invoice->paid) {
                $invoices[] = $invoice;
            }
        }

        return $invoices;
    }
}
```

Heck, let's over-achieve by adding some PHPDoc that shows that this method return an array of \Stripe\Invoice objects:

225 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 202
    /**
     * @param User $user
     * @return \Stripe\Invoice[]
     */
    public function findPaidInvoices(User $user)
    {
... lines 209 - 222
    }
}
```

Listing all Invoices

Thanks to this function, on the account page, at the bottom, we'll print a list of all the Customer's invoices. Eventually, they'll be able to click each invoice to see *all* the details.

Start in ProfileController... all the way at the top: this method renders the account page. Fetch the invoices with $invoices = $this->get('stripe_client')->findPaidInvoices() with the current user. Pass that as a new variable into the template:

179 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 19
    public function accountAction()
    {
... lines 22 - 35
        $invoices = $this->get('stripe_client')
            ->findPaidInvoices($this->getUser());

        return $this->render('profile/account.html.twig', [
... lines 40 - 44
            'invoices' => $invoices
        ]);
    }
... lines 48 - 177
}
```

Now inside the template, find the bottom of the table. Add a new row, and title it Invoices:

180 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 67
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 73 - 88
                <table class="table">
                    <tbody>
... lines 91 - 148
                        <tr>
                            <th>Invoices</th>
                            <td>
... lines 152 - 160
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 166 - 174
        </div>
    </div>
</div>
{% endblock %}
... lines 179 - 180
```

Next, create a list and then loop with for invoice in invoices. Add the endfor. Create an anchor tag, but keep the href empty for now - we don't have an invoice "show" page yet. Add some classes to make this look a little fancy:

180 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 67
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 73 - 88
                <table class="table">
                    <tbody>
... lines 91 - 148
                        <tr>
                            <th>Invoices</th>
                            <td>
                                <div class="list-group">
                                    {% for invoice in invoices %}
                                        <a href="" class="list-group-item">
... lines 155 - 157
                                        </a>
                                    {% endfor %}
                                </div>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 166 - 174
        </div>
    </div>
</div>
{% endblock %}
... lines 179 - 180
```

So let's see, the user might be looking for a *specific* invoice, so let's print its date. Check out the Invoice API. Hey! There's actually a date field, which is a UNIX timestamp. Print invoice.date and then pipe that through the date filter with Y-m-d:

180 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 67
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 73 - 88
                <table class="table">
                    <tbody>
... lines 91 - 148
                        <tr>
                            <th>Invoices</th>
                            <td>
                                <div class="list-group">
                                    {% for invoice in invoices %}
                                        <a href="" class="list-group-item">
                                            Date: {{ invoice.date|date('Y-m-d') }}
... lines 156 - 157
                                        </a>
                                    {% endfor %}
                                </div>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 166 - 174
        </div>
    </div>
</div>
{% endblock %}
... lines 179 - 180
```

Next, add a span label, float it right, and inside, add the amount: $ then print invoice.amount_due / 100 to convert it from cents to dollars:

180 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 67
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 73 - 88
                <table class="table">
                    <tbody>
... lines 91 - 148
                        <tr>
                            <th>Invoices</th>
                            <td>
                                <div class="list-group">
                                    {% for invoice in invoices %}
                                        <a href="" class="list-group-item">
                                            Date: {{ invoice.date|date('Y-m-d') }}

                                            <span class="label label-success pull-right">${{ invoice.amount_due/100 }} </span>
                                        </a>
                                    {% endfor %}
                                </div>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 166 - 174
        </div>
    </div>
</div>
{% endblock %}
... lines 179 - 180
```

The amount_due field is what the user should have *actually* been charged, after accounting for coupons or a positive account balance.

Try things out so far: head back to the account page and refresh! Bam! Here's our long invoice list. Next, let's give each invoice its own detailed display page, complete with invoice items, discounts and anything else that might have happened on that invoice.

Chapter 35: Displaying All the Invoice Details

To see *all* the details for each invoice, we need an "invoice show" page. Head to ProfileController and add a new method for this: showInvoiceAction(). Give it a URL: /profile/invoices/{invoiceId}. And a name: account_invoice_show, and then add the $invoiceId argument:

192 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 178
    /**
     * @Route("/profile/invoices/{invoiceId}", name="account_invoice_show")
     */
    public function showInvoiceAction($invoiceId)
    {
... lines 184 - 189
    }
}
```

Before doing anything else, copy that route name, head back to the account template and fill in the href by printing path() then pasting the route name. For the second argument, pass in the wildcard: invoiceId set to invoice.id, which will be the Stripe invoice ID:

180 lines | app/Resources/views/profile/account.html.twig

```
... lines 1 - 67
{% block body %}
<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 73 - 88
                <table class="table">
                    <tbody>
... lines 91 - 148
                        <tr>
                            <th>Invoices</th>
                            <td>
                                <div class="list-group">
                                    {% for invoice in invoices %}
                                        <a href="{{ path('account_invoice_show', {invoiceId: invoice.id}) }}" class="list-group-item">
                                            Date: {{ invoice.date|date('Y-m-d') }}

                                            <span class="label label-success pull-right">${{ invoice.amount_due/100 }} </span>
                                        </a>
                                    {% endfor %}
                                </div>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
... lines 166 - 174
        </div>
    </div>
</div>
{% endblock %}
... lines 179 - 180
```

Fetch One Invoice's Data

Back in the controller, our work here is pretty simple: we'll just ask Stripe for this *one* Invoice. In StripeClient, we don't have a method that returns just *one* invoice, so let's add one: public function findInvoice() with an $invoiceId argument. Inside, return the elegant \Stripe\Invoice::retrieve($invoiceId):

230 lines | src/AppBundle/StripeClient.php

```
... lines 1 - 8
class StripeClient
{
... lines 11 - 224
    public function findInvoice($invoiceId)
    {
        return \Stripe\Invoice::retrieve($invoiceId);
    }
}
```

Love it!

In the controller, use this: $stripeInvoice = $this->get('stripe_client')->findInvoice() with $invoiceId:

192 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 181
    public function showInvoiceAction($invoiceId)
    {
        $stripeInvoice = $this->get('stripe_client')
            ->findInvoice($invoiceId);
... lines 186 - 189
    }
}
```

To make things really nice, you'll probably want to wrap this in a try-catch block: if there's a 404 error from Stripe, you'll want to catch that exception and throw the normal $this->createNotFoundException(). That'll cause our site to return a 404 error, instead of 500 error.

Finally, render a new template: how about profile/invoice.html.twig. Pass this an invoice variable set to $stripeInvoice:

192 lines | src/AppBundle/Controller/ProfileController.php

```
... lines 1 - 14
class ProfileController extends BaseController
{
... lines 17 - 181
    public function showInvoiceAction($invoiceId)
    {
        $stripeInvoice = $this->get('stripe_client')
            ->findInvoice($invoiceId);

        return $this->render('profile/invoice.html.twig', array(
            'invoice' => $stripeInvoice
        ));
    }
}
```

Rendering Invoice Details

Instead of creating that template by hand, let's take a shortcut. If you downloaded the "start" code from the site, you should have a tutorial/ directory with an invoice.html.twig file inside. Copy that and paste it into your profile/ templates directory:

79 lines | app/Resources/views/profile/invoice.html.twig

```
{% extends 'base.html.twig' %}
{% import _self as macros %}

{% macro currency(rawStripeAmount) %}
    {% if rawStripeAmount < 0 %}-{% endif %}${{ (rawStripeAmount/100)|abs }}
{% endmacro %}

{% block body %}

<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
                <h1>Invoice {{ invoice.date|date('Y-m-d') }}</h1>

                <table class="table">
```

```
<table class="table">
  <thead>
    <tr>
      <th>To</th>
      {# or put company information here #}
      <th>{{ app.user.email }}</th>
    </tr>
    <tr>
      <th>Invoice Number</th>
      <th>
        {{ invoice.id }}
      </th>
    </tr>
  </thead>
</table>

<table class="table table-striped">
  <tbody>
    {% if invoice.starting_balance %}
      <tr>
        <td>Starting Balance</td>
        <td>
          {{ macros.currency(invoice.starting_balance) }}
        </td>
      </tr>
    {% endif %}
    {% for lineItem in invoice.lines.data %}
      <tr>
        <td>
          {% if lineItem.description %}
            {{ lineItem.description }}
          {% elseif (lineItem.plan) %}
            Subscription to {{ lineItem.plan.name }}
          {% endif %}
        </td>
        <td>
          {{ macros.currency(lineItem.amount) }}
        </td>
      </tr>
    {% endfor %}

    {% if invoice.discount %}
      <tr>
        <td>Discount: {{ invoice.discount.coupon.id }}</td>
        <td>
          {{ macros.currency(invoice.discount.coupon.amount_off * -1) }}
        </td>
      </tr>
    {% endif %}

    <tr>
      <th>Total</th>
      <th>
        {{ macros.currency(invoice.amount_due) }}
      </th>
    </tr>
```

```
                </tbody>
            </table>
        </div>
    </div>
</div>
{% endblock %}
```

Before we look deeper at this, let's make sure it works! Refresh the profile page, then click into one of the discounted invoices. Score! It's got all the important stuff: the subscription, the discount and the total at the bottom.

Here's the deal: there is an infinite number of ways to render an invoice. But the tricky part is understanding all the different pieces that you need to include. Let's take a look at invoice.html.twig to see what it's doing.

The Components of an Invoice

First, the Invoice has a starting_balance field:

79 lines | app/Resources/views/profile/invoice.html.twig

```
... lines 1 - 7
{% block body %}

<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 14 - 31
                <table class="table table-striped">
                    <tbody>
                        {% if invoice.starting_balance %}
                            <tr>
                                <td>Starting Balance</td>
                                <td>
                                    {{ macros.currency(invoice.starting_balance) }}
                                </td>
                            </tr>
                        {% endif %}
... lines 42 - 71
                    </tbody>
                </table>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

This answers the question: "how much was the customer's account balance before charging this invoice?" If the balance is positive, then this was used to *discount* the invoice before charging the customer. By printing it here, it'll help explain the total.

Tip

It's possible that not *all* of the Customer's balance was used. You could also use the ending_balance field to check.

Second, since each charge is a line item, we can loop through each one and print its details. But, each line item *might* be for an individual product *or* for a subscription. It's a little weird, but I've found that the best way to handle this is to check to see if lineItem.description is set:

79 lines | app/Resources/views/profile/invoice.html.twig

```
... lines 1 - 7
{% block body %}

<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 14 - 31
                <table class="table table-striped">
                    <tbody>
... lines 34 - 41
                        {% for lineItem in invoice.lines.data %}
                            <tr>
                                <td>
                                    {% if lineItem.description %}
                                        {{ lineItem.description }}
                                    {% elseif (lineItem.plan) %}
                                        Subscription to {{ lineItem.plan.name }}
                                    {% endif %}
                                </td>
                                <td>
                                    {{ macros.currency(lineItem.amount) }}
                                </td>
                            </tr>
                        {% endfor %}
... lines 56 - 71
                    </tbody>
                </table>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

If it *is* set, then print it. In that case, this line item is either an individual product - in which case the description is the product's name - *or* it's a proration subscription line item that's created when a user changes between plans. In that case, the description is really nice: it explains exactly what this charge or credit means.

But if the description is blank, this is for a normal subscription charge. Print out "Subscription to" and then lineItem.plan.name.

In both cases, for the amount, print lineItem.amount. Oh, that macros.currency() thing is a macro I setup that helps manage negative numbers and adds the $ sign:

79 lines | app/Resources/views/profile/invoice.html.twig

```
... line 1
{% import _self as macros %}

{% macro currency(rawStripeAmount) %}
    {% if rawStripeAmount < 0 %}-{% endif %}${{ (rawStripeAmount/100)|abs }}
{% endmacro %}
... lines 7 - 79
```

After the line items, there's just *one* more thing to worry about: discounts! We already know that you can create Coupons and attach them to a Customer at checkout. When a Coupon has been used, it's known as a "discount" on the invoice. Let's print the coupon's ID and the amount off thanks to the coupon:

79 lines | app/Resources/views/profile/invoice.html.twig

```
... lines 1 - 7
{% block body %}

<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 14 - 31
                <table class="table table-striped">
                    <tbody>
... lines 34 - 56
                        {% if invoice.discount %}
                            <tr>
                                <td>Discount: {{ invoice.discount.coupon.id }}</td>
                                <td>
                                    {{ macros.currency(invoice.discount.coupon.amount_off * -1) }}
                                </td>
                            </tr>
                        {% endif %}
... lines 65 - 71
                    </tbody>
                </table>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

If you want to support Coupons for both a set amount off *and* a percentage off, you'll need to do a little bit more work here.

Finally, at the bottom: print the total by using the amount_due field. After taking everything above into account, this should be the amount they were charged:

79 lines | app/Resources/views/profile/invoice.html.twig

```
... lines 1 - 7
{% block body %}

<div class="nav-space">
    <div class="container">
        <div class="row">
            <div class="col-xs-6">
... lines 14 - 31
                <table class="table table-striped">
                    <tbody>
... lines 34 - 65
                        <tr>
                            <th>Total</th>
                            <th>
                                {{ macros.currency(invoice.amount_due) }}
                            </th>
                        </tr>
                    </tbody>
                </table>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

So... Store Invoices Locally?

Ok! Once you know which fields to render, it's not too bad. But this approach has one big downside: we don't have any of the invoice data in *our* database: we're relying on a third party to store everything. It also means that it'll be a little bit harder to query or report on invoice data. And finally, the invoice pages may load a little slow, since we're waiting for an API request to Stripe to finish.

If you *do* want to store invoices locally, it's not too much more work. Of course, you'll need an invoices table with whatever columns are important for you to store: like the amount charged, and maybe some discount details.

That's simple enough, but how and when would we populate this? Webhooks! Specifically, the invoice.created webhook: just respond to this and create a "copy" in your database of whatever info you want. You'll also want to listen to invoice.updated to catch any *changes* to an invoice, like when it goes from unpaid to paid.

If that's important to you, go for it!

Ahhhh, now we *really* did it! We've made it to the end. This stuff is tough, but you should feel empowered. Creating a payment system is about more than just accepting credit cards, it's about giving your customers a great, bug-free, surprise-free and joyful experience. Go out there and make that a reality!

And as always, if you have any questions, ask us in the comments.

Seeya guys next time!