An essential building block of a successful marketplace is to process payments on behalf of others. Moreover, you want your users to list products and get paid instantly through your market. In this setting, you take a commission on each transaction. Stripe Connect is an ideal solution to make this happen.

The process to setup such a framework is far from simple. While Stripe is probably the easiest solution out there, there are a lot of elements and pitfalls to tackle before you can have a working function.

I have therefore written this tutorial to give you a jump start and guide you through the complete setup. My estimate is that a novice developer would need about 1 hour to complete all the steps.

You can unlock the complete tutorial and source code by downloading this component. The package contains all you need to enable your users to accept money on behalf of others, while you subsequently collect fees in the process. The source code is provided in *NodeJS* and comes with an extensive documentation. This post can be therefore seen as a preview as it leaves out the code examples and the source code.

Important: this tutorial is a follow up on Stripe Charge, which comes with an elaborative documentation on how to setup Stripe and your servers. If you are not already familiar with processing payments using Stripe, then please complete these instructions before proceeding with this tutorial.

Getting Started

The first step is to Register your platform on Stripe, which you can do by following the url: `https://dashboard.stripe.com/account/applications/settings`, and pressing on **Connect** at the top. In this tutorial we will be working with a **Standalone Account**.

Sidenote on Standalone Accounts versus Managed Accounts

From the docs: *A standalone Stripe account is a normal Stripe account that is controlled directly by the account holder (i.e. your platform's user). A user with standalone account will have a relationship with Stripe, be able to log in to the dashboard, can process charges on their own, and can disconnect their account from your platform.*.

This is definitely the recommended option if you are just starting out as it will significantly decrease your development process. To use the alternative, a **Managed Account**, you will need to implement many interactions, such as collecting all the information that Stripe needs. You might be interested in this option if you wish to make Stripe completely invisible to the account holder, but that is a tutorial on its own.

Redirect URIs and Constants

An essential field in registering your Standalone Account is the Redirect URIs (define as `REDIRECT_URI`), which we will talk about later. This is essentially the field that Stripe uses to handle the callbacks after your user has authenticated their Stripe Application.

If you have setup Stripe before using Stripe Charge, then you are already familiar with the setup. In that case, you can use your existing `SERVER_SIDE_URL` and add `/oauth/callback` at the end. If you have previously hosted your server-side on Heroku, then the final `REDIRECT_URI` would look something like:

```
[your-heroku-app-name].herokuapp.com/oauth/callback
```

Add this url to both the development as production Redirect URIs.

If you have not setup the server-side before, then please follow the instructions in the section **Preparing the Server-Side** from the Stripe Charge template.

Extending our Server-Side

Now that we have activated Stripe Connect, we need to extend our server-side code to cope with authenticating users and asking them for permission to connect their Stripe Account to your platform.

The first step is to add the `CLIENT_ID` (use development or production depending on yourother settings) to your server-side code, which you host on your `SERVER_SIDE_URL`. Basically add the following line of code at the top of your page:

```
var CLIENT_ID   = 'ca_<YOUR-CLIENT-ID>';
```

Retrieving the Authorization Credentials

Next is to extend our server-side to retrieve the Authorization Credentials, which are basically the parameters that are used when your user is receiving a payment on behalf of your platform. Let's start by defining two new urls to authorize the user and to retrieve a token for retrieving the authorization credentials:

```
var TOKEN_URI     = 'https://connect.stripe.com/oauth/token';
var AUTHORIZE_URI = 'https://connect.stripe.com/oauth/authorize';
```

We proceed by creating an authorization route which can be called from the client-side, by adding this line of code:

```
router.get('/authorize',

function(req, res) {
  console.log("--------- authorize")
  console.log("authorize: firebase userId or custom field: ", req.query.userId)
  console.log("authorize: firebase auth token: ", req.query.token)

  // optional: session storage (see full source code for description)

  res.redirect(AUTHORIZE_URI + '?' + qs.stringify({
    response_type: 'code',
    scope: 'read_write',
    client_id: CLIENT_ID
  }));
})
```

Then let's extend our routes by adding the callback (i.e. `REDIRECT_URI`) which will capture the `AUTHORIZATION_CODE`, which gives our platform the permission to connect the users Stripe Account and retrieve the Authorization Credentials. We do this as follows:

```
router.get('/oauth/callback', function(req, res) {

  console.log("--------- callback")
  console.log("get auth code: ", req.query.code)
  console.log("get userId: ", session["cookie"].userId)

  var AUTHORIZATION_CODE        = req.query.code;

  // Optional Firebase Session code here (see complete source code)

  // Make /oauth/token endpoint POST request
  request.post({
    url: TOKEN_URI,
    form: {
      grant_type:'authorization_code',
      client_id: CLIENT_ID,
      code: AUTHORIZATION_CODE,
```

```
      client_secret: STRIPE_API_SECRET_KEY
    }
  }, function(err, r, body) {
    var SCData = JSON.parse(body); // Authorization Credentials
    if(err) {
      res.send("Oops something went wrong");
    } else {
      // <--
      res.send("Success! Your account is now setup to use Stripe Connect. You may
close this window.");
    };

    // Alternatively, save it securely on Firebase here. See full source code for
details

  });
})
```

Store the Authorization Credentials in your Database

The data of interest in this part is the object `SCData`, the **Authorization Credentials**, on which we apply `JSON.parse()` such that we can **store it in our noSQL database**. In the complete source code I give an example of how to do this securely with Firebase, as it also requires retrieving a Firebase Authentication Token.

Now that our server-side is pre-configured, push the latest changes to your server such that we can consume it from the client-side.

Client-side setup

It's time to implement our changes on the client-side. This is just a matter of redirecting the user to an appropriate route which we have defined on the server side. Let's define the following:

```
//app.js
var STRIPE_URL_AUTHORIZE            = SERVER_SIDE_URL + "/authorize";
```

Define this url in your **controller** as follows:

```
// controllers-account.js
$scope.DynamicStripeAuthorizeUrl = STRIPE_URL_AUTHORIZE;
```

This can be now called from your view by creating a link:

```
// account.html
<a
  href="{{DynamicStripeAuthorizeUrl}}" target="_blank"
  ng-show="!status.loadingAuthToken">
```

```
  Connect with Stripe
</a>
```

Firebase Token

Read this part only if you are using Firebase as your back-end. In order to store the Authorized Credentials of the user securely, it is recommended and required that these credentials are written to your database on the server. This means that your user somehow needs to be authenticated to your database provider on your server as well. When using Firebase, a good option is to generate authentication tokens from the client-side and pass these in `STRIPE_URL_AUTHORIZE`, which is called when the users attempts to connect their account to Stripe Connect. We can do this by extending the url as follows:

```
// controllers-account.js
$scope.DynamicStripeAuthorizeUrl = STRIPE_URL_AUTHORIZE + "?userId=" + AuthData.uid
+ "&token=" + fbAuthToken;
```

Here `AuthData` is the object retrieved from monitoring the authentication state of the user. You can read more about it on the Firebase website: In our setting, we generate `fbAuthToken` when the user enters a certain view, for which he needs to be authenticated already. I recommend doing this in the Account Settings of the user, where you also put the **Connect to Stripe button**. Note that I added the `ng-show=...` caption to make sure that the link is only shown when the token is actually loaded.

```
// controllers-account.js
$scope.status = {loadingAuthToken: false};
function loadWhenEnteringView() {
  $scope.status["loadingAuthToken"] = true;
  FirebaseCheckout.generateFBAuthToken(account.AuthData.uid).then(
    function(fbAuthToken){
        // update the dynamic url for stripe connect
        $scope.DynamicStripeAuthorizeUrl = STRIPE_URL_AUTHORIZE +
            "?userId=" + account.AuthData.uid + "&token=" + fbAuthToken;
        $scope.status["loadingAuthToken"]    = false;
    },
    function(error){
      console.log(error);
    };
  );
};
```

with `generateFBAuthToken()` defined as a function in the service `FirebaseCheckout`:

```
// services-firebase-utils.js
// inject $q as a dependency

var STRIPE_FIREBASE_GEN_TOKEN  = STRIPE_SERVER_URL + "/firebase/generatetoken";
```

```
self.generateFBAuthToken = function(userId) {
  var qGen = $q.defer();
  //console.log("requesting to: " + STRIPE_FIREBASE_GEN_TOKEN);
  $http.post(STRIPE_FIREBASE_GEN_TOKEN, {userId: userId})
  .success(
      function(fbAuthToken){
        if(fbAuthToken != null) {
            qGen.resolve(fbAuthToken);
        } else {
            qGen.reject("ERROR_NULL");
        }
      }
  )
  .error(
      function(error){
          qGen.reject(error);
      }
  );
  return qGen.promise;
};
```

Here we are making a call to the url `STRIPE_FIREBASE_GEN_TOKEN` which is another route on our server side. I have included this part of the code in the full source code.

Now that you have succesfully generated a token, you can use this parameter on your client-side to authenticate the users and thus write the Authorized Credentials, stored in the object `SCData` (see above), to your database. I recommend writing it to a node as follows: `<YOUR-FIREBASE-URL>.firebaseio.com/stripe_connect_data/$userId`, thus seperating it from an existing node that is open for public such as `<YOUR-FIREBASE-URL>.firebaseio.com/users`.

For safetey, extend your firebase security rules as follows:

```
  "stripe_connect_data": {
    "$uid": {
      ".write": "auth !== null && auth.uid === $uid",   // make sure that only the
connected user can write (otherwise someone can put their own account id)
      ".read": true
    }
  }
```

As it is now, everyone can read the `SCData`. We recommend therefore storing only fields that are essential (parsing it on the server-side), such as the destination account id. You could also consider encrypting this data or only providing access during sessions. This is to be covered in another tutorial.

Processing Payments with Stripe Connect

We return again to our server-side as we need to do a final thing to make it possible to actually process the payments made on behalf of your users, and thus to charge a commission (or

'admission fee').

In the Stripe Charge tutorial we processed payments through the route `/charge` which we accessed by sending `$http POST` requests to `STRIPE_URL + "/charge"`. Let's rename this route to /charge/nodestination and then add the following line of code to handle payments with an admission fee:

```
router.post('/charge', function(req, res) {

    console.log("--------- charge")

    console.log("charge: amount:", req.body.stripeAmount)
    console.log("charge: currency:", req.body.stripeCurrency)
    console.log("charge: source:", req.body.stripeSource)
    console.log("charge: description", req.body.stripeDescription)
    console.log("charge: destination", req.body.stripeDestinationAccountId)
    console.log("charge: noodlio fee", req.body.noodlioApplicationFee)

    var charge = stripe.charges.create(
      {
        amount:           req.body.stripeAmount,      // amount in cents, again
        currency:         req.body.stripeCurrency,
        source:           req.body.stripeSource,
        description:      req.body.stripeDescription,
        application_fee:  req.body.noodlioApplicationFee
      },
      {stripe_account: req.body.stripeDestinationAccountId},
      function(err, charge) {
        // callback
        if(err) {
          console.log("charge callback: error:", err)
          res.json(err);
        } else {
          // <--
          console.log("charge callback: charge:", charge)
          res.json(charge);
        }
      }
    ) // ./ charge
});
```

As you see we have two additional parameters, the `noodlioApplicationFee` expressed in cents, and `stripeDestinationAccountId`.The destination account id is one of the fields from the object `SCData` which you stored in the node `stripe_connect_data/$userId` before.

On the client-side, we need to change the way that we charge our clients in the controllers and services. A payment flow becomes then as follows:

- validate the credit card details and the `stripeToken` using Stripe Checkout (see Stripe Charge tutorial)
- retrieve the `stripeDestinationAccountId` from `stripe_connect_data/$userId`

- send a `$http POST` request with the additional parameters to the updated `STRIPE_URL + "/charge"`

This corresponds to the following changes in our services:

```
// app.js

var STRIPE_URL_CHARGE               = STRIPE_SERVER_URL + "/charge";
var STRIPE_URL_CHARGE_NODESTINATION = STRIPE_SERVER_URL + "/charge/nodestination";
var ADMISSION_FEE                   = 0.1;
```

```
// services-payments.js

self.chargeUser = function(priceUSD, productUserId, stripeToken) {
  var qCharge = $q.defer();

  var chargeServerUrl = null;

  var curlData = {
      stripeCurrency:         "usd",
      stripeAmount:           Math.floor(priceUSD*100),
      stripeSource:           stripeToken,
      stripeDescription:      "Payment on behalf of the user"
  };

  // retrieve the destination account details of the owner of the product
  self.getStripeConnectAuth_value(productUserId).then(
    function(SCData){

      if(SCData != null) {

        // extend the curl data if the user has connected its account
        curlData["stripeDestinationAccountId"]  = SCData.stripe_user_id;
        curlData["noodlioApplicationFee"] =
calculateNoodlioApplicationFee(priceUSD);

        chargeServerUrl                         = STRIPE_URL_CHARGE;

      } else {

        // optional: if not, then transfer the full amount to the platform
        // note, you might want to change this as the user might feel screwed
        // thus add: qCharge.reject("STRIPE_CONNECT_NULL");
        chargeServerUrl = STRIPE_URL_CHARGE_NODESTINATION

      };

      // -->
      proceedCharge()
    },
    function(nosuccess){
      console.log(nosuccess)
      qCharge.reject(nosuccess);
    }
```

```
  );

  // process the payment and retrieve the invoice data
  function proceedCharge() {
    $http.post(chargeServerUrl, curlData)
    .success(
      function(StripeInvoiceData){
        if(StripeInvoiceData != null && StripeInvoiceData.hasOwnProperty('status'))
{
          if(StripeInvoiceData.status == "succeeded") {
           qCharge.resolve(StripeInvoiceData);
          }
          qCharge.reject(StripeInvoiceData);
        } else {
          qCharge.reject(StripeInvoiceData);
        }
      }
    )
    .error(
      function(error){
        console.log(error)
        qCharge.reject(error);
      }
    );
  };
  return qCharge.promise;
};


function calculateNoodlioApplicationFee(netPrice) {
  var appFee = netPrice * (ADMISSION_FEE);
  return Math.floor(appFee);
};
```

The only missing parameter in this example is `stripeToken`. You can get the source code of this function in the Stripe Charge starter in the file `services.js`.

NodeJS Source code

Available for download here

Questions or Feedback

That is all, you have now a working marketplace with Stripe Connect. If you have questions or issues, feel free to drop an email to noodlio@seipel-ibisevic.com, write/read your comments on Noodlio, or on this website.