FREE eBook

# LEARNING PayPal

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#paypal

# Table of Contents

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: paypal

It is an unofficial and free PayPal ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PayPal.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Chapter 1: Getting started with PayPal

## Remarks

These guides will take the user through account setup procedures for applications, accounts, etc. It will contain everything that is needed for working with the PayPal APIs.

## Versions

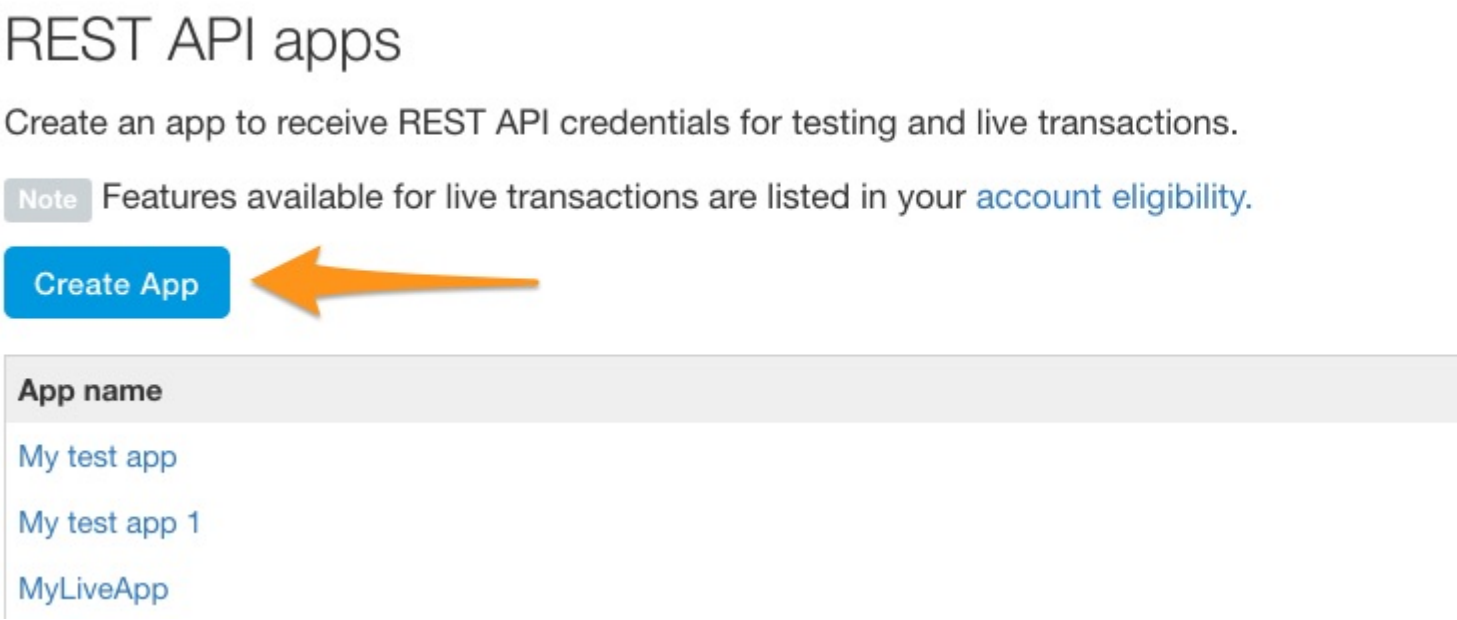
| Version | Release Date |
| --- | --- |
| 1.0.0 | 2016-04-11 |

## Examples

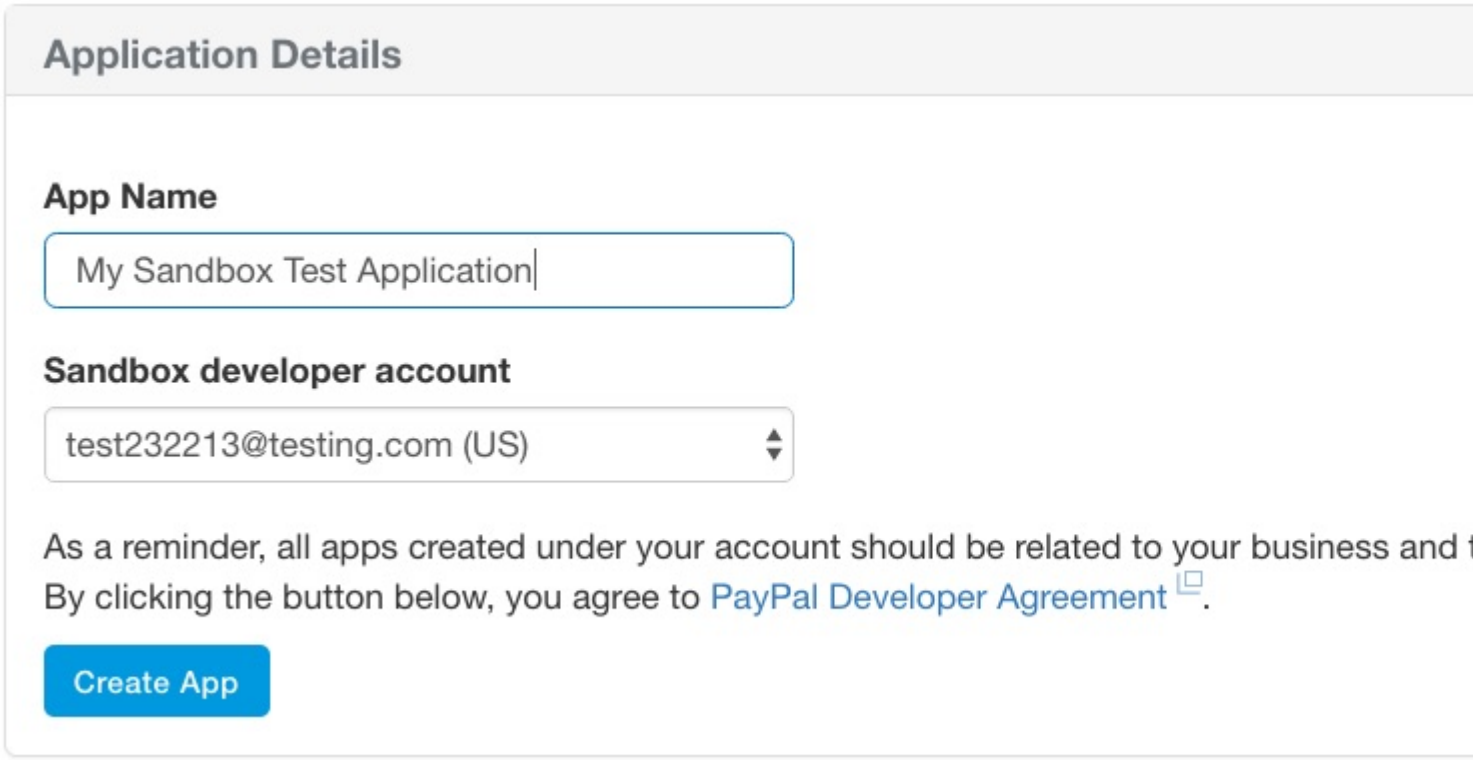
### Creating an application and obtaining client id / secret keys

In order to begin building with PayPal APIs, you have to create an application to obtain a client ID and secret.

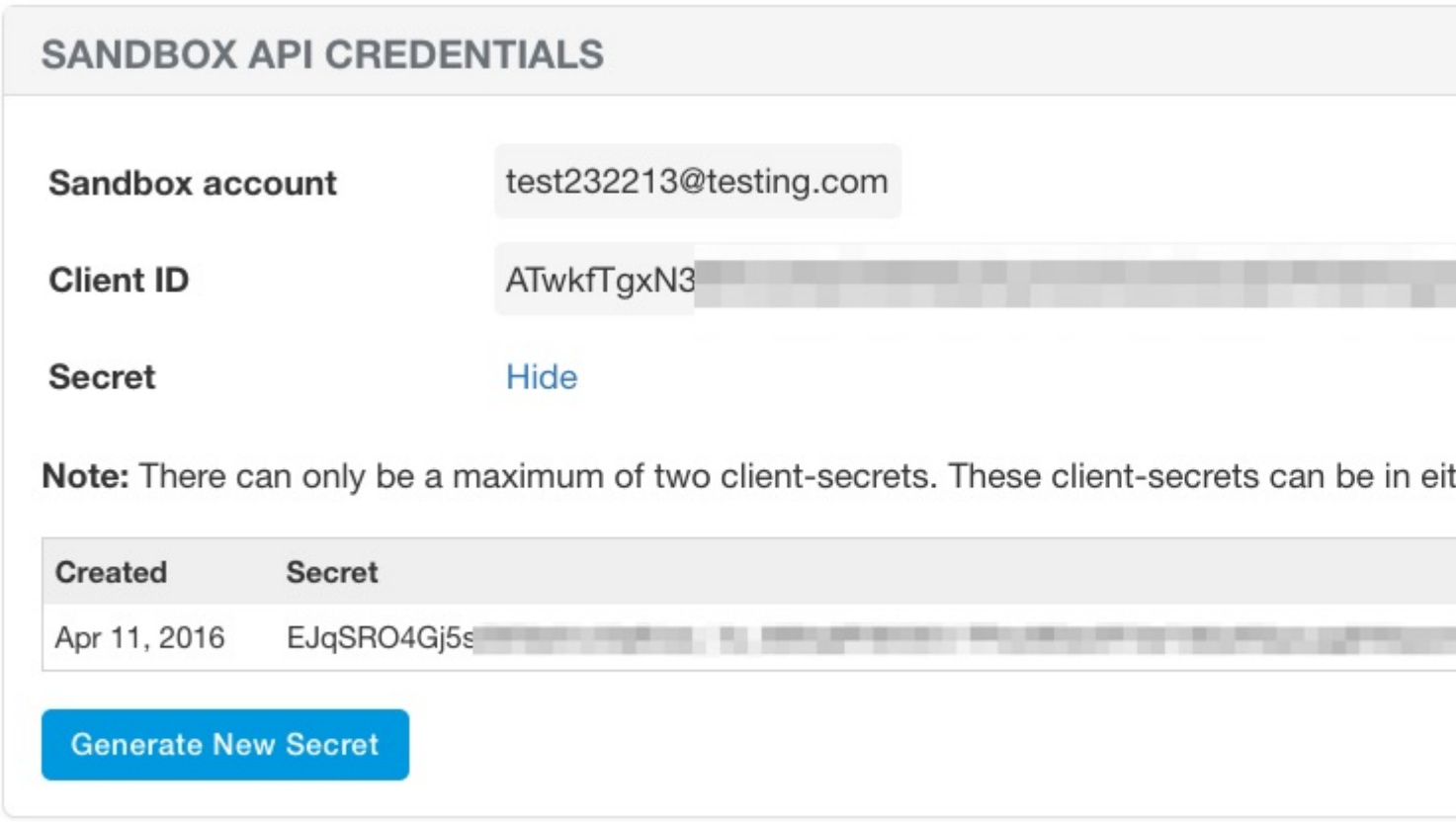
Go to https://developer.paypal.com/developer/applications/, sign in, and click on "Create App", as shown below:




Next, enter an application name, select the sandbox testing account that you want to use (if it's a new account, leave the default value), and click "Create App".




Once the application is created, you will be provided with your sandbox and live client ID and secret, which will look similar to the following:

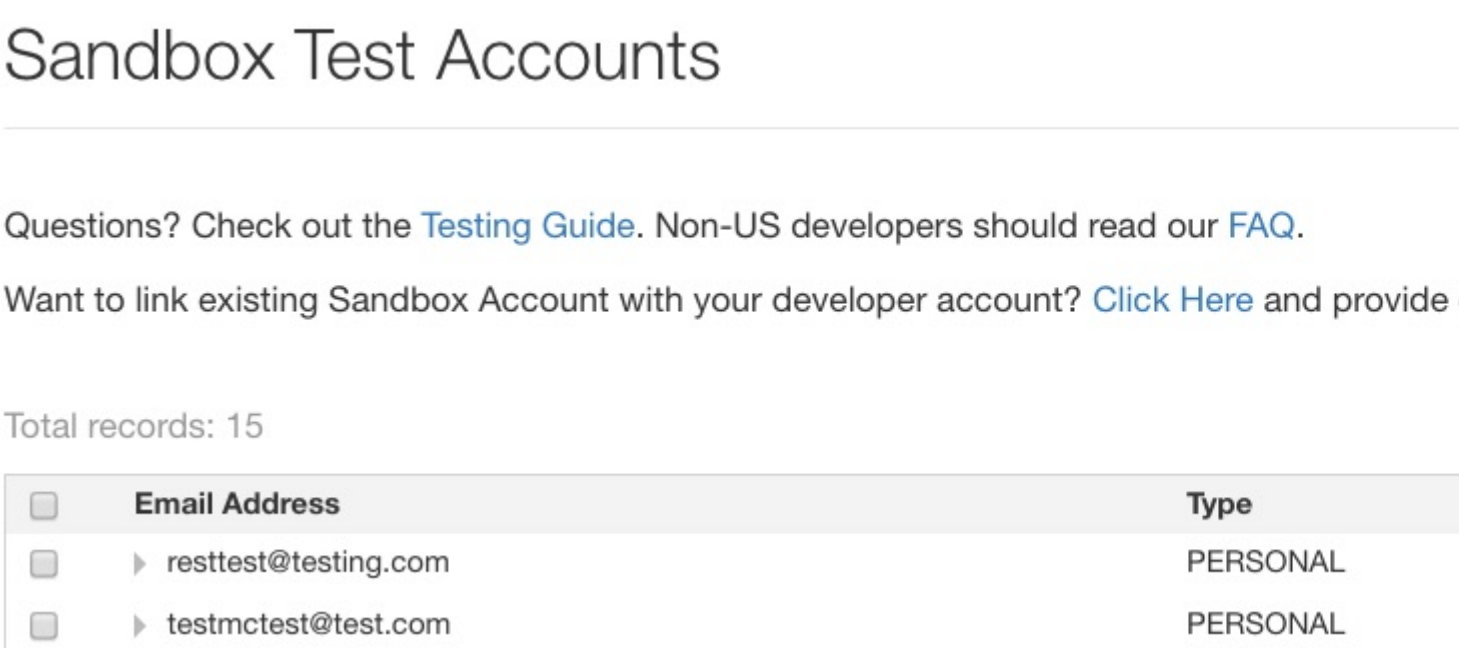



These credentials are what you will use when making requests to PayPal APIs in order to authenticate your application and make requests.

### Setting up sandbox user test accounts

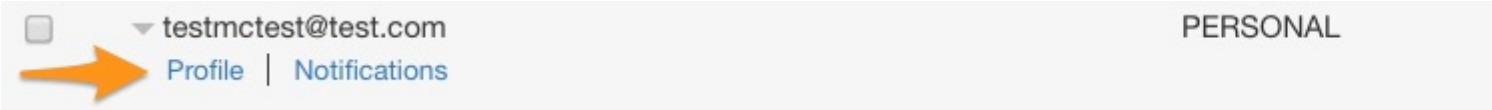
When testing you PayPal integration on sandbox, you'll need to have sandbox user accounts set up to use to go through the payment flow.

Go to https://developer.paypal.com/developer/accounts/, log in using your PayPal account, and click on "Create Account", as below:




Enter in the accounts details for the new test user, including a unique email, account information, payment method, balance, etc, and click on "Create Account" at the bottom of the page once done. This will create the new account for you to begin using.

To see account details for this new user, expand the entry on the accounts page, and click on "Profile".




Once that profile information loads, clicking on the "Funding" tab will give you payment information for that account, including credit card information that may be used for direct credit card processing against sandbox.

NOTE: When using the sandbox API endpoints, you need to use sandbox test account to log in and pay for the test goods, as your live account information will not work.

Read Getting started with PayPal online: https://riptutorial.com/paypal/topic/406/getting-started-with-paypal

# Chapter 2: Creating Subscriptions / Recurring Payments

## Parameters

| Parameter | Details |
|---|---|
| billingAgreementAttributes | Configuration object to create the billing agreement |
| billingPlan | Billing plan ID from the query string |
| billingPlanAttribs | Configuration object to create the billing plan |
| billingPlanUpdateAttributes | Configuration object for changing a billing plan to an active state |
| clientId | Your application client ID (OAuth keys) |
| http | Reference to the http package to set up our simple server |
| isoDate | ISO date for setting the subscription start date |
| links | HATEOAS link object for extracting the redirect URL to PayPal |
| params | Query string parameters |
| paypal | Reference to the PayPal SDK |
| secret | Your application secret (OAuth keys) |
| token | The billing agreement approval token provided after PayPal redirect to execute the billing agreement |

## Remarks

These examples go through the process of creating a subscription / recurring payment system using PayPal.

The process for creating a subscription is to:

1. Create a billing plan. This is a reusable model that outlines the details of the subscription.
2. Activate the billing plan.
3. When you want to create a subscription for a user, you create a billing agreement using the ID of the billing plan that they should be subscribed to.
4. Once created, you redirect the user to PayPal to confirm the subscription. Once confirmed, the user is redirected back to the merchant's website.

5. Lastly, you execute the billing agreement to begin the subscription.

# Examples

### Step 2: Creating a Subscription for a User using a Billing Agreement (Node Sample)

The second step to creating a subscription for a user is to create and execute a billing agreement, based on an existing activated billing plan. This example assumes that you have already gone through and activated a billing plan in the previous example, and have an ID for that billing plan to reference in the example.

When you are setting up a billing agreement to create a subscription for a user, you'll follow 3 steps, which may be reminiscent of processing a PayPal payment:

1. You create a billing agreement, referencing an underlying billing plan via the ID.
2. Once created, you redirect the user to PayPal (if paying via PayPal) to confirm the subscription. Once confirmed, PayPal redirects the user back to your site using the redirect provided in the underlying billing plan.
3. You then execute the billing agreement using a token provided back via the PayPal redirect.

This example is setting up an Express based HTTP server to showcase the billing agreement process.

To start the example, we first need to set up our configuration. We add four requirements, the PayPal SDK, `body-parser` for handling JSON encoded bodies, `http` for our simple server integration, and `express` for the Express framework. We then define our client ID and secret from creating an application, configure the SDK for the sandbox, then configure bodyParser for handling JSON bodies.

```
var paypal = require('paypal-rest-sdk'),
    bodyParser = require('body-parser'),
    http = require('http'),
    app = require('express')();

var clientId = 'YOUR APPLICATION CLIENT ID';
var secret = 'YOUR APPLICATION SECRET';

paypal.configure({
  'mode': 'sandbox', //sandbox or live
  'client_id': clientId,
  'client_secret': secret
});

app.use(bodyParser.json());
```

Our first step in the billing agreement is to create a route to handle the creation of a billing agreement, and redirecting the user to PayPal to confirm that subscription. We are assuming that a billing plan ID is passed as a query string parameter, such as by loading the following URL with a plan ID from the previous example:

```
http://localhost:3000/createagreement?plan=P-3N543779E9831025ECYGDNVQ
```

We now need to use that information to create the billing agreement.

```
app.get('/createagreement', function(req, res){
    var billingPlan = req.query.plan;

    var isoDate = new Date();
    isoDate.setSeconds(isoDate.getSeconds() + 4);
    isoDate.toISOString().slice(0, 19) + 'Z';

    var billingAgreementAttributes = {
        "name": "Standard Membership",
        "description": "Food of the World Club Standard Membership",
        "start_date": isoDate,
        "plan": {
            "id": billingPlan
        },
        "payer": {
            "payment_method": "paypal"
        },
        "shipping_address": {
            "line1": "W 34th St",
            "city": "New York",
            "state": "NY",
            "postal_code": "10001",
            "country_code": "US"
        }
    };

    // Use activated billing plan to create agreement
    paypal.billingAgreement.create(billingAgreementAttributes, function (error,
billingAgreement){
        if (error) {
            console.error(error);
            throw error;
        } else {
            //capture HATEOAS links
            var links = {};
            billingAgreement.links.forEach(function(linkObj){
                links[linkObj.rel] = {
                    'href': linkObj.href,
                    'method': linkObj.method
                };
            })

            //if redirect url present, redirect user
            if (links.hasOwnProperty('approval_url')){
                res.redirect(links['approval_url'].href);
            } else {
                console.error('no redirect URI present');
            }
        }
    });
});
```

We start by extracting the billing plan ID from the query string and create the date when the plan should start.

The next object definition, `billingAgreementAttributes`, consists of information for the subscription. It contains readable information on the plan, a reference to the billing plan ID, the payment method, and shipping details (if needed for the subscription).

Next, a call to `billingAgreement.create(...)` is made, passing in the `billingAgreementAttributes` object we just created. If all is successful, we should have a billing agreement object passed back to us containing details about our newly created subscription. That object also contains a number of HATEOAS links providing us next steps that can be taken on this newly created agreement. The one we care about here is labeled as `approval_url`.

We loop through all provided links to put them into an easily referenced object. If `approval_url` is one of those links, we redirect the user to that link, which is PayPal.

At this point the user confirms the subscription on PayPal, and is redirected back to the URL provided in the underlying billing plan. Along with that URL, PayPal will also pass a token along the query string. That token is what we're going to use to execute (or start) the subscription.

Let's set up that functionality in the following route.

```
app.get('/processagreement', function(req, res){
    var token = req.query.token;

    paypal.billingAgreement.execute(token, {}, function (error, billingAgreement) {
        if (error) {
            console.error(error);
            throw error;
        } else {
            console.log(JSON.stringify(billingAgreement));
            res.send('Billing Agreement Created Successfully');
        }
    });
});
```

We extract the token from the query string, then make a call to `billingAgreement.execute`, passing along that token. If all is successful, we now have a valid subscription for the user. The return object contains information about the active billing agreement.

Lastly, we set up our HTTP server to listen for traffic to our routes.

```
//create server
http.createServer(app).listen(3000, function () {
   console.log('Server started: Listening on port 3000');
});
```

## Step 1: Creating a Subscription Model using a Billing Plan (Node Sample)

When creating a subscription for a user, you first need to create and activate a billing plan that a user is then subscribed to using a billing agreement. The full process for creating a subscription is detailed in the remarks of this topic.

Within this example, we're going to be using the PayPal Node SDK. You can obtain it from NPM

using the following command:

```
npm install paypal-rest-sdk
```

Within our .js file, we first set up our SDK configuration, which includes adding a requirement for the SDK, defining our client ID and secret from creating our application, and then configuring the SDK for the sandbox environment.

```
var paypal = require('paypal-rest-sdk');

var clientId = 'YOUR CLIENT ID';
var secret = 'YOUR SECRET';

paypal.configure({
  'mode': 'sandbox', //sandbox or live
  'client_id': clientId,
  'client_secret': secret
});
```

Next, we need to set up two JSON objects. The `billingPlanAttribs` object contains the information and payment breakdown for the billing plan that we can subscribe users to, and the `billingPlanUpdateAttributes` object contains values for setting the billing plan to an active state, allowing it to be used.

```
var billingPlanAttribs = {
    "name": "Food of the World Club Membership: Standard",
    "description": "Monthly plan for getting the t-shirt of the month.",
    "type": "fixed",
    "payment_definitions": [{
        "name": "Standard Plan",
        "type": "REGULAR",
        "frequency_interval": "1",
        "frequency": "MONTH",
        "cycles": "11",
        "amount": {
            "currency": "USD",
            "value": "19.99"
        }
    }],
    "merchant_preferences": {
        "setup_fee": {
            "currency": "USD",
            "value": "1"
        },
        "cancel_url": "http://localhost:3000/cancel",
        "return_url": "http://localhost:3000/processagreement",
        "max_fail_attempts": "0",
        "auto_bill_amount": "YES",
        "initial_fail_amount_action": "CONTINUE"
    }
};

var billingPlanUpdateAttributes = [{
    "op": "replace",
    "path": "/",
    "value": {
```

```
        "state": "ACTIVE"
    }
}];
```

Within the `billingPlanAttribs` object, there are some relevant pieces of information:

- **name / description / type**: Basic visual information to describe the plan, and the type of plan.
- **payment_definitions**: Information on how the plan should function and be billed. More details on fields here.
- **merchant_preferences**: Additional fee structures, redirect URLs, and settings for the subscription plan. More details on fields here.

With those objects in place, we can now create and activate the billing plan.

```
paypal.billingPlan.create(billingPlanAttribs, function (error, billingPlan){
    if (error){
        console.log(error);
        throw error;
    } else {
        // Activate the plan by changing status to Active
        paypal.billingPlan.update(billingPlan.id, billingPlanUpdateAttributes, function(error,
response){
            if (error) {
                console.log(error);
                throw error;
            } else {
                console.log(billingPlan.id);
            }
        });
    }
});
```

We call `billingPlan.create(...)`, passing in the `billingPlanAttribs` object that we just created. If that is successful, the return object will contain information about the billing plan. For the sake of the example, we just need to use the billing plan ID in order to activate the plan for use.

Next, we call `billingPlan.update(...)`, passing in the billing plan ID and the `billingPlanUpdateAttributes` object we created earlier. If that is successful, our billing plan is now active and ready to use.

In order to create a subscription for a user (or multiple users) on this plan, we'll need to reference the billing plan id (`billingPlan.id` above), so store that in a place that can be referenced easily.

In the second subscription step, we need to create a billing agreement based on the plan we just created and execute it to begin processing subscriptions for a user.

Read Creating Subscriptions / Recurring Payments online:
https://riptutorial.com/paypal/topic/467/creating-subscriptions---recurring-payments

# Chapter 3: Making a Credit Card Payment (Node)

## Parameters

| Parameter | Details |
|---|---|
| card_data | JSON object containing payment information for transaction |
| credit_card_details | JSON object containing credit card data that is sent to PayPal to be vaulted |
| client_id | Your PayPal application client ID (OAuth 2 credentials) |
| paypal | PayPal Node SDK reference |
| secret | Your PayPal application secret (OAuth 2 credentials) |
| uuid | Reference to the node-uuid package |

## Remarks

This sample takes the user through crediting a simple credit card transaction using the PayPal SDKs.

## Examples

### Node Sample

Start by installing the PayPal Node module from NPM

```
npm install paypal-rest-sdk
```

In your application file, add in the configuration information for the SDK

```
var paypal = require('paypal-rest-sdk');

var client_id = 'YOUR CLIENT ID';
var secret = 'YOUR SECRET';

paypal.configure({
    'mode': 'sandbox', //sandbox or live
    'client_id': client_id,
    'client_secret': secret
});
```

We add the requirement for the SDK, then set up variables for the client ID and secret that were obtained when creating an application. We then configure our application using these details, and specify the environment that we are working in (live or sandbox).

Next, we set up the JSON object that contains the payment information for the payer.

```
var card_data = {
  "intent": "sale",
  "payer": {
    "payment_method": "credit_card",
    "funding_instruments": [{
      "credit_card": {
        "type": "visa",
        "number": "4417119669820331",
        "expire_month": "11",
        "expire_year": "2018",
        "cvv2": "874",
        "first_name": "Joe",
        "last_name": "Shopper",
        "billing_address": {
          "line1": "52 N Main ST",
          "city": "Johnstown",
          "state": "OH",
          "postal_code": "43210",
          "country_code": "US" }}}]},
  "transactions": [{
    "amount": {
      "total": "7.47",
      "currency": "USD",
      "details": {
        "subtotal": "7.41",
        "tax": "0.03",
        "shipping": "0.03"}},
    "description": "This is the payment transaction description."
}]};
```

Add an `intent` of `sale`, and a `payment_method` of `credit_card`. Next, add in the card and address details for the credit card under `funding_instruments`, and the amount to be charged under `transactions`. Multiple transaction objects can be placed here.

Lastly, we make a request to `payment.create(...)`, passing in our `card_data` object, in order to process the payment.

```
paypal.payment.create(card_data, function(error, payment){
  if(error){
    console.error(error);
  } else {
    console.log(payment);
  }
});
```

If the transaction was successful, we should see a response object similar to the following:

```
{
  "id": "PAY-9BS08892W3794812YK4HKFQY",
  "create_time": "2016-04-13T19:49:23Z",
```

```
  "update_time": "2016-04-13T19:50:07Z",
  "state": "approved",
  "intent": "sale",
  "payer": {
    "payment_method": "credit_card",
    "funding_instruments": [
      {
        "credit_card": {
          "type": "visa",
          "number": "xxxxxxxxxxxx0331",
          "expire_month": "11",
          "expire_year": "2018",
          "first_name": "Joe",
          "last_name": "Shopper",
          "billing_address": {
            "line1": "52 N Main ST",
            "city": "Johnstown",
            "state": "OH",
            "postal_code": "43210",
            "country_code": "US"
          }
        }
      }
    ]
  },
  "transactions": [
    {
      "amount": {
        "total": "7.47",
        "currency": "USD",
        "details": {
          "subtotal": "7.41",
          "tax": "0.03",
          "shipping": "0.03"
        }
      },
      "description": "This is the payment transaction description.",
      "related_resources": [
        {
          "sale": {
            "id": "0LB81696PP288253D",
            "create_time": "2016-04-13T19:49:23Z",
            "update_time": "2016-04-13T19:50:07Z",
            "amount": {
              "total": "7.47",
              "currency": "USD"
            },
            "state": "completed",
            "parent_payment": "PAY-9BS08892W3794812YK4HKFQY",
            "links": [
              {
                "href":
"https:\/\/api.sandbox.paypal.com\/v1\/payments\/sale\/0LB81696PP288253D",
                "rel": "self",
                "method": "GET"
              },
              {
                "href":
"https:\/\/api.sandbox.paypal.com\/v1\/payments\/sale\/0LB81696PP288253D\/refund",
                "rel": "refund",
                "method": "POST"
```

```
              },
              {
                "href": "https:\/\/api.sandbox.paypal.com\/v1\/payments\/payment\/PAY-
9BS08892W3794812YK4HKFQY",
                "rel": "parent_payment",
                "method": "GET"
              }
            ],
            "fmf_details": {

            },
            "processor_response": {
              "avs_code": "X",
              "cvv_code": "M"
            }
          }
        }
      ]
    }
  ],
  "links": [
    {
      "href": "https:\/\/api.sandbox.paypal.com\/v1\/payments\/payment\/PAY-
9BS08892W3794812YK4HKFQY",
      "rel": "self",
      "method": "GET"
    }
  ],
  "httpStatusCode": 201
}
```

In this return object, having a `state` of `approved` tells us that the transaction was successful. Under the `links` object are a number of HATEOAS links that provide potential next steps that can be taken on the action that was just performed. In this case, we can retrieve information about the payment by making a GET request to the `self` endpoint provided.

## Making a Payment with a Vaulted Credit Card (Node)

In this example, we'll be looking at how to store a credit card using the PayPal vault, then reference that stored credit card to process a credit card transaction for a user.

The reason why we would want to use the vault is so that we don't have to store sensitive credit card information on our own servers. We simply reference the payment method via a provided vault ID, meaning that we don't have to deal with many PCI compliance regulations with storing the credit cards ourselves.

As with previous samples, we start with setting up our environment.

```
var paypal = require('paypal-rest-sdk'),
    uuid = require('node-uuid');

var client_id = 'YOUR CLIENT ID';
var secret = 'YOUR SECRET';

paypal.configure({
  'mode': 'sandbox', //sandbox or live
```

```
  'client_id': client_id,
  'client_secret': secret
});
```

The one difference to previous samples here is that we are requiring a new package, `node-uuid`, which is to be used to generate unique UUID's for the payers when storing the card. You can install that package via:

```
npm install node-uuid
```

Next, we define the credit card JSON object that will be sent to the PayPal vault for storage. It contains information from the card, as well as a unique payer ID that we generate using `node-uuid`. You should store this unique `payer_id` in your own database as it will be used when creating a payment with the vaulted card.

```
var create_card_details = {
    "type": "visa",
    "number": "4417119669820331",
    "expire_month": "11",
    "expire_year": "2018",
    "first_name": "John",
    "last_name": "Doe",
    "payer_id": uuid.v4()
};
```

Lastly, we need to store the credit card and process the payment using that card. To vault a credit card, we call `credit_card.create(...)`, passing in the `credit_card_details` object that we just created. If all goes well, we should have an object returned to us with details about the vaulted card. For the sake of a payment with that card, we only really need two pieces of information: the payer_id that we already stored, and the vault ID, that should also be stored as a reference in our own database.

```
paypal.credit_card.create(create_card_details, function(error, credit_card){
    if(error){
        console.error(error);
    } else {
        var card_data = {
            "intent": "sale",
            "payer": {
                "payment_method": "credit_card",
                "funding_instruments": [{
                    "credit_card_token": {
                        "credit_card_id": credit_card.id,
                        "payer_id": credit_card.payer_id
                    }
                }]
            },
            "transactions": [{
                "amount": {
                    "total": "7.47",
                    "currency": "USD",
                    "details": {
                        "subtotal": "7.41",
                        "tax": "0.03",
```

```
                        "shipping": "0.03"
                    }
                },
                "description": "This is the payment transaction description."
            }]
        };

        paypal.payment.create(card_data, function(error, payment){
            if(error){
                console.error(error);
            } else {
                console.log(JSON.stringify(payment));
            }
        });
    }
});
```

In the section following the successful vaulting of the credit card, we are simply defining the card details and processing the payment, as we did with the previous credit card processing example. The main difference in the structure of the `card_data` object is the `funding_instruments` section, that we define under `payer`. Instead of defining the credit card information, we instead use the following object that contains the vault ID reference, and the payer ID:

```
"credit_card_token": {
    "credit_card_id": credit_card.id,
    "payer_id": credit_card.payer_id
}
```

That is how we use a vaulted card to process a payment.

Read Making a Credit Card Payment (Node) online:
https://riptutorial.com/paypal/topic/444/making-a-credit-card-payment--node-

# Chapter 4: Making a PayPal payment

## Parameters

| Parameter | Details |
|---|---|
| clientId | Your PayPal application client ID (OAuth 2 credentials) |
| links | Simple reference object for all return HATEOAS links from PayPal |
| paymentId | The ID of the payment returned from PayPal in order to complete payment |
| payerId | The ID of the payer returned from PayPal in order to complete payment |
| paypal | PayPal Node SDK reference |
| payReq | JSON object containing payment information for transaction |
| req | The request object from the server request |
| res | The response object from the server request |
| secret | Your PayPal application secret (OAuth 2 credentials) |

## Remarks

These samples cover how to process a payment via PayPal, using the PayPal SDKs. These are simple request samples that outline the multi-step process for allowing this payment option.

## Examples

### Node Express Server Example

In this example, we're going to set up an Express server integration to display how to process a payment with PayPal, using the PayPal Node SDK. We will use a static JSON structure for the payment details for the sake of brevity.

There are three general steps that we will follow when building out the functions to handle the PayPal payment:

1. We create a JSON object containing the payment that we intend to process through PayPal. We then send that to PayPal to obtain a link to redirect the user to in order to confirm payment.
2. Next, we redirect the user to PayPal to confirm the payment. Once confirmed, PayPal redirects the user back to our application.

3. Once returned to the app, we complete the payment on behalf of the user.

Breaking this down as a simple Node app, we start by obtaining the PayPal Node SDK from NPM:

```
npm install paypal-rest-sdk
```

Next, we set up the app configuration and packages.

```
var http = require('http'),
    paypal = require('paypal-rest-sdk'),
    bodyParser = require('body-parser'),
    app = require('express')();

var client_id = 'YOUR APPLICATION CLIENT ID';
var secret = 'YOUR APPLICATION SECRET';

//allow parsing of JSON bodies
app.use(bodyParser.json());

//configure for sandbox environment
paypal.configure({
    'mode': 'sandbox', //sandbox or live
    'client_id': client_id,
    'client_secret': secret
});
```

We require four requirements for this app:

1. The HTTP package for our server.
2. The PayPal Node SDK package.
3. The bodyParser package for working with JSON encoded bodies.
4. The Express framework for our server.

The next few lines set up variables for the client ID and secret that were obtained when creating an application. We then set up `bodyParser` to allow for JSON encoded bodies, then configure our application using the application details, and specify the environment that we are working in (live for production or sandbox for testing).

Now let's create the route for creating a payment request with PayPal.

```
app.get('/create', function(req, res){
    //build PayPal payment request
    var payReq = JSON.stringify({
        'intent':'sale',
        'redirect_urls':{
            'return_url':'http://localhost:3000/process',
            'cancel_url':'http://localhost:3000/cancel'
        },
        'payer':{
            'payment_method':'paypal'
        },
        'transactions':[{
            'amount':{
                'total':'7.47',
                'currency':'USD'
```

```
            },
            'description':'This is the payment transaction description.'
        }]
    });

    paypal.payment.create(payReq, function(error, payment){
        if(error){
            console.error(error);
        } else {
            //capture HATEOAS links
            var links = {};
            payment.links.forEach(function(linkObj){
                links[linkObj.rel] = {
                    'href': linkObj.href,
                    'method': linkObj.method
                };
            })

            //if redirect url present, redirect user
            if (links.hasOwnProperty('approval_url')){
                res.redirect(links['approval_url'].href);
            } else {
                console.error('no redirect URI present');
            }
        }
    });
});
```

The first thing we do is set up the payment request JSON object, which contains the information that we need to provide PayPal with to create the payment. We set the `intent` to `sale`, specify the redirect URLs (where PayPal should forward the user to after they confirm / cancel the payment), add in a `payment_method` of `paypal` to signal that we will make a PayPal payment, then specify the transaction information for the payer to confirm.

We then call `payment.create(...)`, passing in our `payReq` object. This will send the create payment request to PayPal. Once that returns, and is successful, we can loop through the provided HATEOAS links in the return object to extract the URL that we need to redirect the user to, which is labeled under `approval_url`.

The format for the HATEOAS links can cause fragile reference code if used directly, so we loop through all provided links and put them in a better reference object to future proof against changes. If the `approval_url` is then found in that object, we redirect the user.

At this point the user is redirected to PayPal to confirm the payment. Once they do, they are redirected back to the `return_url` that we specified in the `createPayment(...)` function.

We now have to provide a route to handle that return, in order to complete the payment.

```
app.get('/process', function(req, res){
    var paymentId = req.query.paymentId;
    var payerId = { 'payer_id': req.query.PayerID };

    paypal.payment.execute(paymentId, payerId, function(error, payment){
        if(error){
            console.error(error);
```

```
        } else {
            if (payment.state == 'approved'){
                res.send('payment completed successfully');
            } else {
                res.send('payment not successful');
            }
        }
    });
});
```

When the user is returned back to your app, there will be three query string parameters that will be sent along as well, the `paymentId`, `PayerID`, and `token`. We only need to deal with the first two.

We extract the parameters, and place the PayerID in a simple object for the need of the payment execution step. Next, a call is made to `payment.execute(...)`, passing in those two parameters, in order to complete the payment.

Once that request is made, we see if the payment completed successfully by checking if `payment.state` is set to `approved`. If so, we can store what we need from the payment object that is returned.

Our last step is to initialize our server and listen for traffic coming to the routes we specified.

```
//create server
http.createServer(app).listen(3000, function () {
   console.log('Server started: Listening on port 3000');
});
```

Once the server is initialized, going to `http://localhost:3000/create` initializes the payment process.

Read Making a PayPal payment online: https://riptutorial.com/paypal/topic/449/making-a-paypal-payment

# Chapter 5: Mobile Future Payments (End to End App)

## Remarks

This example shows a practical end to end example of creating a PayPal future payment from an Android device, using a Node server.

## Examples

### Android Step 1: Layout, Initialization, and Handling Server Response

The complete sample code for this application (Android + Node server) is available in the PayPal Developer Github repository.

The first stage of creating the Android portion of our application is to set up a basic layout and handle responses that come back from the server that we'll set up in Node.

Start by creating a new PayPalConfiguration object to house your application information.

```
private static PayPalConfiguration config = new PayPalConfiguration()
        .environment(PayPalConfiguration.ENVIRONMENT_SANDBOX)
        .clientId("YOUR APPLICATION CLIENT ID")
        .merchantName("My Store")
        .merchantPrivacyPolicyUri(Uri.parse("https://www.example.com/privacy"))
        .merchantUserAgreementUri(Uri.parse("https://www.example.com/legal"));
```

Next, we add a simple button to `onCreate(...)` to act as our payment initiation. This is simply to trigger off the action, and should be placed as the initiation process for creating a future payment for a user (e.g. when they agree upon a subscription).

```
@Override
protected void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    final Button button = (Button) findViewById(R.id.paypal_button);
}
```

Under `res > layout > activity_main.xml` we add the definition for the button with its associated action, when clicked it calls `beginFuturePayment(...)`, which we'll define in a minute.

```
<Button android:id="@+id/paypal_button"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:text="@string/paypal_button"
    android:onClick="beginFuturePayment" />
```

Under `res > values > strings.xml` we then add a string reference for the button.

```
<string name="paypal_button">Process Future Payment</string>
```

Now we add the button handler, to initiate the call to begin the future payment process when the user clicks the button. What we are doing here is starting the payment service with the configuration object we set up at the top of this example.

```
public void beginFuturePayment(View view){
    Intent serviceConfig = new Intent(this, PayPalService.class);
    serviceConfig.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    startService(serviceConfig);

    Intent intent = new Intent(this, PayPalFuturePaymentActivity.class);
    intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    startActivityForResult(intent, 0);
}
```

When that call to make a future payment is initiated, we will be given some information that will need to be sent to our server. We extract this information from the valid future payment request ( `authCode` and `metadataId`), then make execute the async request to the server to complete the future payment (detailed in step 2).

```
@Override
protected void onActivityResult (int requestCode, int resultCode, Intent data){
    if (resultCode == Activity.RESULT_OK){
        PayPalAuthorization auth =
data.getParcelableExtra(PayPalFuturePaymentActivity.EXTRA_RESULT_AUTHORIZATION);
        if (auth != null){
            try{
                //prepare params to be sent to server
                String authCode = auth.getAuthorizationCode();
                String metadataId = PayPalConfiguration.getClientMetadataId(this);
                String [] params = {authCode, metadataId};

                //process async server request for token + payment
                ServerRequest req = new ServerRequest();
                req.execute(params);

            } catch (JSONException e) {
                Log.e("FPSample", "JSON Exception: ", e);
            }
        }
    } else if (resultCode == Activity.RESULT_CANCELED) {
        Log.i("FPSample", "User canceled.");
    } else if (resultCode == PayPalFuturePaymentActivity.RESULT_EXTRAS_INVALID) {
        Log.i("FPSample", "Invalid configuration");
    }
}
```

Lastly, we define our `onDestroy()`.

```
@Override
public void onDestroy(){
    stopService(new Intent(this, PayPalService.class));
```

```
    super.onDestroy();
}
```

## Android Step 2: Async Server Request

The complete sample code for this application (Android + Node server) is available in the PayPal Developer Github repository.

At this point the PayPal future payments button has been clicked, we have an auth code and metadata ID from the PayPal SDK, and we need to pass those on to our server to complete the future payment process.

In the background process below, we are doing a few things:

- We set up the URI that for our server to be `http://10.0.2.2:3000/fpstore`, which is hitting the `/fpstore` endpoint of our server running on localhost.
- The JSON object that will be sent through is then set up, which contains the auth code and metadata ID.
- The connection is then made. In the case of a successful request (200 / 201 range) we can expect a response back from the server. We read that response and then return it.
- Lastly, we have a `onPostExecute(...)` method set up to handle that returned server string. In the case of this example, it's simply logged.

```
public class ServerRequest extends AsyncTask<String, Void, String> {
    protected String doInBackground(String[] params){
        HttpURLConnection connection = null;
        try{
            //set connection to connect to /fpstore on localhost
            URL u = new URL("http://10.0.2.2:3000/fpstore");
            connection = (HttpURLConnection) u.openConnection();
            connection.setRequestMethod("POST");

            //set configuration details
            connection.setRequestProperty("Content-Type", "application/json");
            connection.setRequestProperty("Accept", "application/json");
            connection.setAllowUserInteraction(false);
            connection.setConnectTimeout(10000);
            connection.setReadTimeout(10000);

            //set server post data needed for obtaining access token
            String json = "{\"code\": \"" + params[0] + "\", \"metadataId\": \"" + params[1] +
"\"}";
            Log.i("JSON string", json);

            //set content length and config details
            connection.setRequestProperty("Content-length", json.getBytes().length + "");
            connection.setDoInput(true);
            connection.setDoOutput(true);
            connection.setUseCaches(false);

            //send json as request body
            OutputStream outputStream = connection.getOutputStream();
            outputStream.write(json.getBytes("UTF-8"));
            outputStream.close();
```

```
            //connect to server
            connection.connect();

            //look for 200/201 status code for received data from server
            int status = connection.getResponseCode();
            switch (status){
                case 200:
                case 201:
                    //read in results sent from the server
                    BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                    StringBuilder sb = new StringBuilder();
                    String line;
                    while ((line = bufferedReader.readLine()) != null){
                        sb.append(line + "\n");
                    }
                    bufferedReader.close();

                    //return received string
                    return sb.toString();
            }

        } catch (MalformedURLException ex) {
            Log.e("HTTP Client Error", ex.toString());
        } catch (IOException ex) {
            Log.e("HTTP Client Error", ex.toString());
        } catch (Exception ex) {
            Log.e("HTTP Client Error", ex.toString());
        } finally {
            if (connection != null) {
                try{
                    connection.disconnect();
                } catch (Exception ex) {
                    Log.e("HTTP Client Error", ex.toString());
                }
            }
        }
        return null;
    }

    protected void onPostExecute(String message){
        //log values sent from the server - processed payment
        Log.i("HTTP Client", "Received Return: " + message);
    }
}
```

## Android Step 3: Node Server to Get Access Token & Process Payment

The complete sample code for this application (Android + Node server) is available in the PayPal Developer Github repository.

From step 2, an async request has been made to our server at the `/fpstore` endpoint, passing along the auth code and metadata ID. We now need to exchange those for a token in order to complete the request and process the future payment.

First we set up our configuration variables and object.

```
var bodyParser = require('body-parser'),
    http = require('http'),
    paypal = require('paypal-rest-sdk'),
    app = require('express')();

var client_id = 'YOUR APPLICATION CLIENT ID';
var secret = 'YOUR APPLICATION SECRET';

paypal.configure({
    'mode': 'sandbox',
    'client_id': client_id,
    'client_secret': secret
});

app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json());
```

Now we set up an Express route that will listen for POST requests sent to the `/fpstore` endpoint from our Android code.

We are doing a number of things in this route:

- We capture the auth code and metadata ID from the POST body.
- We then make a request to `generateToken()`, passing through the code object. If successful, we obtain a token that can be used to create the payment.
- Next, the config objects are created for the future payment that is to be made, and a request to `payment.create(...)` is made, passing along the future payment and payment config objects. This creates the future payment.

```
app.post('/fpstore', function(req, res){
    var code = {'authorization_code': req.body.code};
    var metadata_id = req.body.metadataId;

    //generate token from provided code
    paypal.generateToken(code, function (error, refresh_token) {
        if (error) {
            console.log(error);
            console.log(error.response);
        } else {
            //create future payments config
            var fp_config = {'client_metadata_id': metadata_id, 'refresh_token':
refresh_token};

            //payment details
            var payment_config = {
                "intent": "sale",
                "payer": {
                    "payment_method": "paypal"
                },
                "transactions": [{
                    "amount": {
                        "currency": "USD",
                        "total": "3.50"
                    },
                    "description": "Mesozoic era monster toy"
                }]
            };
```

```
            //process future payment
            paypal.payment.create(payment_config, fp_config, function (error, payment) {
                if (error) {
                    console.log(error.response);
                    throw error;
                } else {
                    console.log("Create Payment Response");
                    console.log(payment);

                    //send payment object back to mobile
                    res.send(JSON.stringify(payment));
                }
            });
        }
    });
});
```

Lastly, we create the server on to listen on port 3000.

```
//create server
http.createServer(app).listen(3000, function () {
   console.log('Server started: Listening on port 3000');
});
```

Read Mobile Future Payments (End to End App) online:
https://riptutorial.com/paypal/topic/4537/mobile-future-payments--end-to-end-app-

# Chapter 6: Mobile PayPal / Credit Card Payments

## Parameters

| Parameter | Details |
|---|---|
| button | Simple payment button |
| config | PayPal configuration object housing our client ID (from application creation) and the environment we want to use (sandbox or live) |
| payment | PayPal payment details |
| paymentConfig | Configuration Intent for the payment information and settings |
| serviceConfig | Configuration Intent for the config parameter data |

## Remarks

Samples related to processing payments on mobile devices

## Examples

### Android: Accepting a PayPal / Credit Card Payment

In this tutorial we're going to learn how to set up the PayPal Android SDK to process a simple payment via either a PayPal payment or a credit card purchase. At the end of this example, you should have a simple button in an application that, when clicked, will forward the user to PayPal to confirm a set payment, then return the user back to the application and log the confirmation of payment.

The complete application code for this example is available in the PayPal Developer Github Repository.

Let's get started.

The first step is to obtain and add the SDK to your project. We add the reference to our build.gradle dependancies like so:

```
dependencies {
    compile 'com.paypal.sdk:paypal-android-sdk:2.14.1'
    ...
}
```

Now we head over to our MainActivity.java file (or wherever you'd like to add the PayPal button integration), and add in a `config` object for our client ID and the environment (sandbox) that we will be using.

```
private static PayPalConfiguration config = new PayPalConfiguration()
    .environment(PayPalConfiguration.ENVIRONMENT_SANDBOX)
    .clientId("YOUR CLIENT ID");
```

Now we're going to create a button in our `onCreate(...)` method, which will enable us to process a payment via PayPal once clicked.

```
@Override
protected void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    final Button button = (Button) findViewById(R.id.paypal_button);
}
```

We now need to define the functionality for that button. In your res > layout > main XML file you can add the following definition for the button, which will define the text and onClick handler for the button with the paypal_button ID.

```
<Button android:id="@+id/paypal_button"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:text="@string/paypal_button"
    android:onClick="beginPayment" />
```

When clicked, the button will call the `beginPayment(...)` method. We can then add the text for the button to our strings.xml file, like so:

```
<string name="paypal_button">Pay with PayPal</string>
```

With the button in place, we now have to handle the button click in order to begin payment processing. Add in the following `beginPayment(...)` method below our previous `onCreate(...)` method.

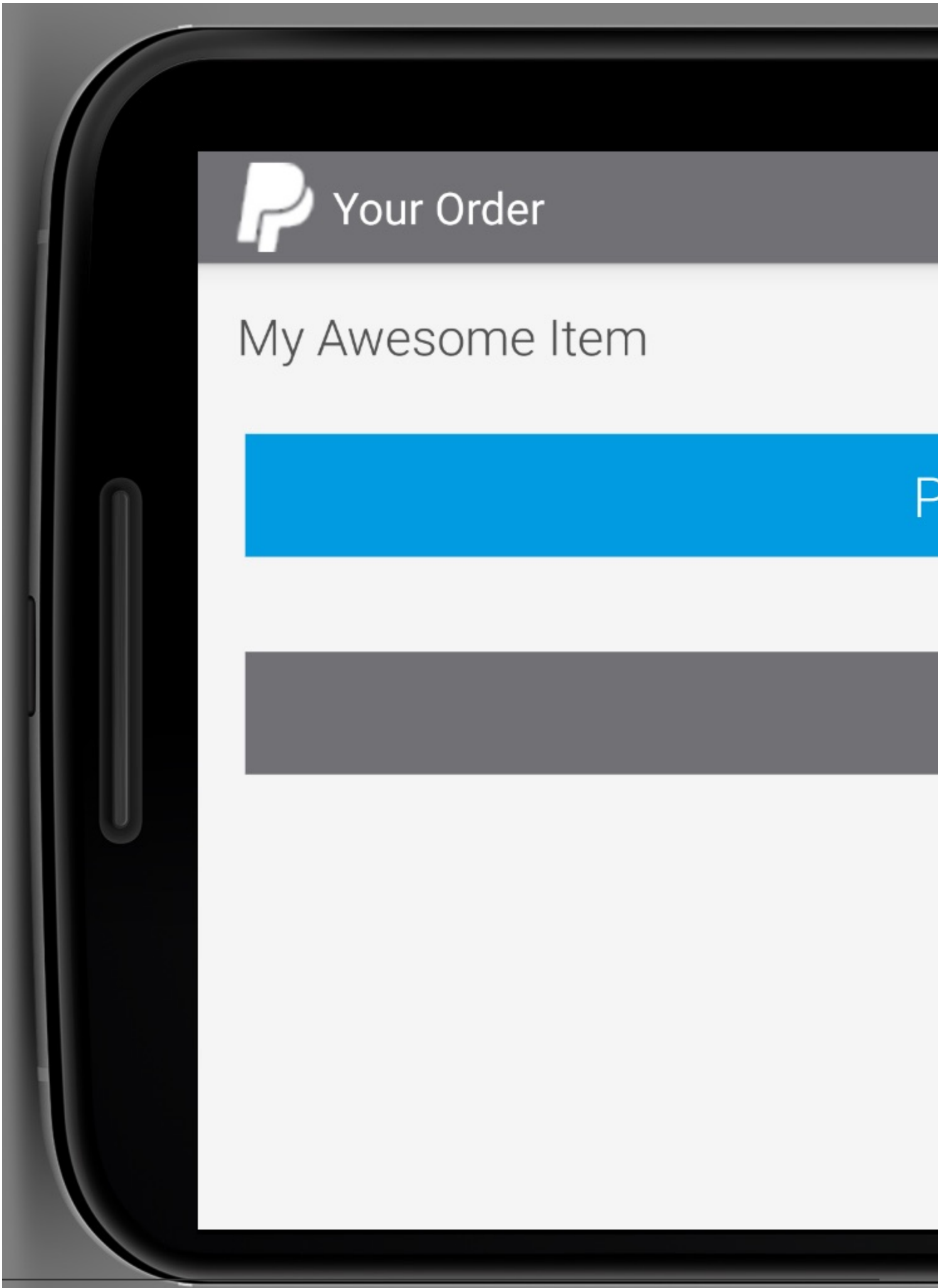
```
public void beginPayment(View view){
    Intent serviceConfig = new Intent(this, PayPalService.class);
    serviceConfig.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    startService(serviceConfig);

    PayPalPayment payment = new PayPalPayment(new BigDecimal("5.65"),
        "USD", "My Awesome Item", PayPalPayment.PAYMENT_INTENT_SALE);

    Intent paymentConfig = new Intent(this, PaymentActivity.class);
    paymentConfig.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    paymentConfig.putExtra(PaymentActivity.EXTRA_PAYMENT, payment);
    startActivityForResult(paymentConfig, 0);
}
```

What we are doing here is first setting up the service intent (`serviceConfig`), using the `config` that we had defined previously for our client ID and the sandbox environment. We then specify the payment object that we want to process. For the sake of this example, we are setting a static price, currency, and description. In your final application, these values should be obtained from what the user is trying to buy in the application. Lastly, we set up the `paymentConfig`, adding in both the `config` and `payment` objects that we had previously defined, and start the activity.

At this point the user will be presented with the PayPal login and payment screens, allowing them to select whether to pay with PayPal or a credit card (via manual entry or card.io if the camera is available). That screen will look something like this:






Once done, we need to have a handler ready for when PayPal forwards the user back to the appl

that comes back is RESULT_OK (user confirmed payment), RESULT_CANCELED (user cancelled payment), or RESULT_EXTRAS_INVALID (there was a configuration issue). In the case of a valid confirmation, we get the object that is returned from the payment and, in this sample, log it. What will be returned to us should look something like the following:

```
{
    "client": {
        "environment": "sandbox",
        "paypal_sdk_version": "2.14.1",
        "platform": "Android",
        "product_name": "PayPal-Android-SDK"
    },
    "response": {
        "create_time": "2016-05-02T15:33:43Z",
        "id": "PAY-0PG63447RB821630KK1TXGTY",
        "intent": "sale",
        "state": "approved"
    },
    "response_type": "payment"
}
```

If we look under the `response` object, we can see that we have a `state` of `approved`, meaning that the payment was confirmed. At this point, that object should be sent to your server to confirm that a payment actually went through. For more information on those steps, see these docs.

Our last step is to cleanup in our `onDestroy(...)`.

```
@Override
public void onDestroy(){
    stopService(new Intent(this, PayPalService.class));
    super.onDestroy();
}
```

That's all there is to it. In this example we've created a simple button to process a payment with either PayPal or a credit card. From this point, there are a few next steps for you to expand upon this sample:

- Pulling in payment information dynamically based on user product selection in the `beginPayment(...)` method.
- Sending the payment confirmation to your server and verifying that the payment actually went through.
- Handling the error and cancellation user cases within the app.

Read Mobile PayPal / Credit Card Payments online:
https://riptutorial.com/paypal/topic/608/mobile-paypal---credit-card-payments

# Chapter 7: Webhooks

## Parameters

| Parameter | Details |
|---|---|
| app | Our Express application reference |
| bodyParser | The body-parser package reference for working with JSON encoded bodies |
| clientId | The application client ID (OAuth 2 credentials) |
| http | The http package for running the server |
| paypal | The PayPal Node SDK reference object |
| secret | The application secret (OAuth 2 credentials) |
| webhookId | ID of the webhook to be modified |
| webhookUpdate | JSON object containing the webhook details to be updated |

## Remarks

These samples cover working examples of how to use PayPal webhooks to provide event monitoring for your application and payments.

## Examples

### Testing Sandbox Webhooks with ngrok and Express (Node)

In this example we're going to look at testing webhook notifications in sandbox, using ngrok to provide a tunnel for our Node HTTP listener, running on localhost, to the internet. For this example, we're going to be using Node to set up notification webhooks for payment events (such as a payment being made), then set up the server to listen for incoming HTTP POST messages from the webhook events.

There are a few steps that we're going to follow here to make this happen:

1. Set up a simple server to listen to incoming POST traffic from the webhooks, which will be the notification from PayPal, and start listening on localhost.
2. Then use ngrok to provide a tunnel from localhost to the internet so that PayPal can post notification through.
3. Lastly, subscribe our application (based on the credentials provided) to webhook events that

we want to track, providing the public ngrok URI from step 2.

**Creating a Webhooks Listener**

The first thing that we need to do is create the listener. The reason why we're starting with the listener is because we need the ngrok live URL to provide to the webhooks when we create or update them.

```
var bodyParser = require('body-parser'),
    http = require('http'),
    app = require('express')();

app.use(bodyParser.json());

app.post('/', function(req, res){
    console.log(JSON.stringify(req.body));
});

//create server
http.createServer(app).listen(3001, function () {
   console.log('Server started: Listening on port 3001');
});
```

Our listener is a simple route using Express. We listen for any incoming POST traffic, then spit out the POST body to the console. We can use this to do whatever we'd like with the listener when it comes in.

When we create the HTTP server at the end, we set it up to listen on localhost port 3001. Run that script now to start listening for traffic.

**Using ngrok to Expose the Listener to the Internet**

With the listener set up on localhost:3001, our next job is to expose that script to the internet, so that traffic can be sent to it, which is the job of ngrok.

Run the following command from a terminal window:

```
ngrok http 3001
```

That will initiate the process of providing a live tunnel for localhost on port 3001, and will provide the following information once run:


As our can see, the live address that we can use to point the PayPal webhook to our running listener on localhost is `http(s)://055b3480.ngrok.io`. That's all we need to know to set up the listener.

**Subscribing to Notifications**

Our last step is to create webhooks for our application, which will create notifications when certain events happen with payments, refunds, etc on our app. We only need to create these webhooks once to bind them to the application, so they do not need to be run each time you want to use them.

First we set up the PayPal environment by adding in the requirement for the PayPal Node SDK, our client ID / secret from creating an application, then configuring the environment for sandbox.

```
var paypal = require('paypal-rest-sdk');

var clientId = 'YOUR APPLICATION CLIENT ID';
var secret = 'YOUR APPLICATION SECRET';

paypal.configure({
  'mode': 'sandbox', //sandbox or live
  'client_id': clientId,
  'client_secret': secret
});
```

Next, we set up the JSON structure for our webhooks. `webhooks` contains two pieces of information, the `url` that all webhook events should be sent to, and the `event_types` that we want to subscribe to.

In the case of this sample, the `url` is set to our ngrok live URL, and the events we are listening for

are cases where payments are completed or denied.

For a complete list of potential events, see https://developer.paypal.com/docs/integration/direct/rest-webhooks-overview/#event-type-support.

Lastly, we pass the `webhooks` object into the call to create the webhooks, `notification.webhook.create`. If successful, PayPal will send notifications to the endpoint we specified, which is running on localhost.

```
var webhooks = {
    "url": "https://436e4d13.ngrok.io",
    "event_types": [{
        "name": "PAYMENT.SALE.COMPLETED"
    },{
        "name": "PAYMENT.SALE.DENIED"
    }
]};

paypal.notification.webhook.create(webhooks, function (err, webhook) {
    if (err) {
        console.log(err.response);
        throw error;
    } else {
        console.log("Create webhook Response");
        console.log(webhook);
    }
});
```

Once we issue a payment using those application credentials, information about the payment state will be sent to the endpoint that we set up.

An example of the POST body that PayPal sends as the notification might looks like the following, which was sent after a successful PayPal payment:

```
{
  "id": "WH-9FE9644311463722U-6TR22899JY792883B",
  "create_time": "2016-04-20T16:51:12Z",
  "resource_type": "sale",
  "event_type": "PAYMENT.SALE.COMPLETED",
  "summary": "Payment completed for $ 7.47 USD",
  "resource": {
    "id": "18169707V5310210W",
    "state": "completed",
    "amount": {
      "total": "7.47",
      "currency": "USD",
      "details": {
        "subtotal": "7.47"
      }
    },
    "payment_mode": "INSTANT_TRANSFER",
    "protection_eligibility": "ELIGIBLE",
    "protection_eligibility_type": "ITEM_NOT_RECEIVED_ELIGIBLE,UNAUTHORIZED_PAYMENT_ELIGIBLE",
    "transaction_fee": {
      "value": "0.52",
      "currency": "USD"
    },
```

```
    "invoice_number": "",
    "custom": "",
    "parent_payment": "PAY-809936371M327284GK4L3FHA",
    "create_time": "2016-04-20T16:47:36Z",
    "update_time": "2016-04-20T16:50:07Z",
    "links": [
      {
        "href": "https:\/\/api.sandbox.paypal.com\/v1\/payments\/sale\/18169707V5310210W",
        "rel": "self",
        "method": "GET"
      },
      {
        "href":
"https:\/\/api.sandbox.paypal.com\/v1\/payments\/sale\/18169707V5310210W\/refund",
        "rel": "refund",
        "method": "POST"
      },
      {
        "href": "https:\/\/api.sandbox.paypal.com\/v1\/payments\/payment\/PAY-
809936371M327284GK4L3FHA",
        "rel": "parent_payment",
        "method": "GET"
      }
    ]
  },
  "links": [
    {
      "href": "https:\/\/api.sandbox.paypal.com\/v1\/notifications\/webhooks-events\/WH-
9FE9644311463722U-6TR22899JY792883B",
      "rel": "self",
      "method": "GET"
    },
    {
      "href": "https:\/\/api.sandbox.paypal.com\/v1\/notifications\/webhooks-events\/WH-
9FE9644311463722U-6TR22899JY792883B\/resend",
      "rel": "resend",
      "method": "POST"
    }
  ]
}
```

## Updating a Webhook with a New URL (Node Sample)

This sample will show you how to update an existing webhook forwarding URL (where the notifications should be POSTed to). To run this, you should have the ID provided back by PayPal when you first created your webhooks.

First, add the PayPal SDK and configure the environment (sandbox below).

```
var paypal = require('paypal-rest-sdk');

var clientId = 'YOUR APPLICATION CLIENT ID';
var secret = 'YOUR APPLICATION SECRET';

paypal.configure({
    'mode': 'sandbox', //sandbox or live
    'client_id': clientId,
    'client_secret': secret
```

```
});
```

Next, set up the JSON structure and webhook details. Assign the ID for your webhook to `webhookId` first. Next, in the `webhookUpdate`, specify an operation of replace, set the `path` to `/url` to specify an update of that resource, and provide the new URL to replace it with under `value`.

```
var webhookId = "YOUR WEBHOOK ID";
var webhookUpdate = [{
    "op": "replace",
    "path": "/url",
    "value": "https://64fb54a2.ngrok.io"
}];
```

Lastly, call `notification.webhook.replace(...)`, passing in `webhookId` and `webhookUpdate`.

paypal.notification.webhook.replace(webhookId, webhookUpdate, function (err, res) { if (err) { console.log(err); throw err; } else { console.log(JSON.stringify(res)); } });

If all succeeds, an object similar to the following should be provided back from PayPal and, in the case of this sample, displayed in the terminal with the newly updated information.

```
{
    "id":"4U496984902512511",
    "url":"https://64fb54a2.ngrok.io",
    "event_types":[{
        "name":"PAYMENT.SALE.DENIED",
        "description":"A sale payment was denied"
    }],
    "links":[{
        "href":"https://api.sandbox.paypal.com/v1/notifications/webhooks/4U496984902512511",
        "rel":"self",
        "method":"GET"
    },{
        "href":"https://api.sandbox.paypal.com/v1/notifications/webhooks/4U496984902512511",
        "rel":"update",
        "method":"PATCH"
    },{
        "href":"https://api.sandbox.paypal.com/v1/notifications/webhooks/4U496984902512511",
        "rel":"delete",
        "method":"DELETE"
    }],
    "httpStatusCode":200
}
```

Read Webhooks online: https://riptutorial.com/paypal/topic/575/webhooks

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with PayPal | Community, Jonathan LeBlanc, Nathan Arthur |
| 2 | Creating Subscriptions / Recurring Payments | Jonathan LeBlanc |
| 3 | Making a Credit Card Payment (Node) | Jonathan LeBlanc |
| 4 | Making a PayPal payment | Jonathan LeBlanc |
| 5 | Mobile Future Payments (End to End App) | Jonathan LeBlanc |
| 6 | Mobile PayPal / Credit Card Payments | Jonathan LeBlanc |
| 7 | Webhooks | Jonathan LeBlanc |