Monetizing Your Application with Payment Flows

2nd Edition

PayPal APIs

Up and Running

O'REILLY®

Matthew A. Russell

PayPal APIs: Up and Running

If your web application's success depends on how quickly and easily users can make transactions, PayPal APIs provide effective solutions you can't afford to overlook. This concise book takes you hands-on through several options to help you determine the best choice for your situation, whether you're collecting money via websites or mobile apps for products and services, donations, or anything else.

In each chapter, you'll work with a different PayPal API by integrating it into the book's sample application, using Python and the Google App Engine framework. This expanded edition introduces two new options: Express Checkout for Digital Goods and Instant Payment Notifications, complete with sample project code. By the end of this book, you'll understand how to take full advantage of PayPal and its powerful features.

- Learn PayPal API basics, and get an introduction to Google App Engine
- Explore the Express Checkout option, and understand what distinguishes it from other generic workflows
- Tailor Express Checkout for electronic documents, videos, and other "in-app" digital purchases
- Apply the Adaptive Payments option for transactions that involve multiple recipients
- Embed the payment process into your site with no mention of PayPal, using Website Payments Pro
- Use the Instant Payment Notifications you receive as triggers to take follow-up action

US $29.99 CAN $31.99
ISBN: 978-1-449-31872-7

O'REILLY®
oreilly.com

SECOND EDITION

PayPal APIs: Up and Running

Matthew A. Russell

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

PayPal APIs: Up and Running, Second Edition
by Matthew A. Russell

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Mary Treseler
Production Editor: Holly Bauer
Proofreader: Holly Bauer
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Revision History for the Second Edition:
2012-04-04 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449318727* for release details.

ISBN: 978-1-449-31872-7

[LSI]

1333467411

Table of Contents

Preface

There has never been a better time to have a keen interest in commerce. The Web has truly accelerated globalization and connected us all through a common network. Information can now be shared at mind-boggling rates, and entrepreneurs everywhere can truly reach a global audience if they're clever (and sometimes lucky) enough to supply the market with what it demands. However, this is old news. Back in the mid-1990s, not long after the Internet officially birthed the Web, buyers and sellers could already transact through eBay, and PayPal soon arrived as the de facto way for money to change hands with the least amount of friction. Fast forward a decade or so, and a lot of exciting things have happened. eBay acquired PayPal back in 2002, and while PayPal continues to be the preferred way to exchange money on eBay, it has since evolved into a powerful platform that offers a vast number of API-based products that allow you to monetize your ideas as seamlessly as possible. If you're interested in tapping into these tremendous possibilities, this book is for you. *As an "Up and Running" title, it doesn't provide complete or exhaustive documentation on all of PayPal's products or even provide very specific direction on handling some of the most common idiosyncrasies that you might encounter*. However, it does aim to present some of the most popular products in fully integrated realistic scenarios with sample project code that you can study and adapt for your particular needs. As the title suggests, this book is designed to get you up and running; it is not a definitive guide.

Each chapter focuses primarily on the topic of integrating commerce payment flows into a reference application that's provided in Appendix A. While one viable approach to demonstrating the integration of PayPal products might have been to introduce a distinct sample application in each chapter, a pragmatic decision to use a single application as a foundation and customize it in various ways according to the content of each chapter was chosen instead. This approach hopefully has the virtues of the sample application being sophisticated enough that it's realistic, fun, and useful, while still allowing each chapter to stand alone and be as atomic and instructive as possible.

Notes About the Second Edition

The first edition version of this book, authored by Michael Balderas, essentially presented a consolidation of PayPal's online documentation and PHP sample code that focused on using the Name-Value Pairs (NVP) APIs for accessing a variety of the most commonly used PayPal products such as Express Checkout, Website Payments Pro, and Adaptive Payments. This edition builds upon that important—albeit fairly abstract—foundation with expanded content, including additional coverage on the exciting new Express Checkout for Digital Goods product and Instant Payment Notifications (IPNs), and includes accompanying sample project code that concretely pulls it all together with a realistic web application. As such, a primary goal of this book is to present PayPal products in a fairly standalone, chapter-by-chapter fashion with the key concepts for integrating each product fully implemented as a sample project. Like any other book, this book tells a coherent (and hopefully enjoyable) story from cover to cover. Although you should be able to skip directly to content of interest with minimal difficulty, you'll get the greatest benefit if you at least skim the entire book before hopping around too much. Appropriate references will be included to any foundational content from previous chapters as needed.

Intended Audience

This book is for any programmer who wants to accept payments for their goods or services through PayPal by using some of PayPal's most popular products. You might be a multimillion-dollar corporation, an individual with an open source project looking to accept donations, a nonprofit requesting donations to help a cause, or a software developer writing mobile apps for cell phones. Regardless, PayPal can provide you with solutions, no matter who you are or how much monetary volume you're processing. The code samples in this book are provided as Python web applications that can be deployed on Google App Engine (GAE) with minimal fuss. Python code is inherently highly readable, and reasonable efforts are made to keep it that way versus using any advanced syntax or nonintuitive Python idioms. Furthermore, the code for the sample web applications has been kept as austere and clear of common Python dependencies—such as Django—as possible so that it is as universally reusable and portable to other languages as possible.

The official Python tutorial is worth perusing if this is your first encounter with Python; however, you really don't need to actually do any Python programming to benefit from this book. The source code and inline comments should be clear enough that it's a fairly trivial exercise for you to port them to your programming platform of choice, and the choice of NVP APIs for PayPal interaction ensures that the payment flows are inherently trivial to understand if you have any programming experience.

How This Book Is Organized

Here is a brief summary of the chapters in the book and what you can expect from each:

Chapter 1, PayPal API Overview
: Provides a 10,000-foot overview of interacting with PayPal APIs as web services and introduces GAE, the primary development platform that's used throughout the book.

Chapter 2, Express Checkout (Including Mobile Express Checkout)
: Showcases Express Checkout, PayPal's premier checkout solution, and demonstrates how to implement a basic Express Checkout payment flow for Tweet Relevance.

Chapter 3, Express Checkout for Digital Goods
: Teaches you how to tailor and improve the Express Checkout flow established in the previous chapter as an Express Checkout for Digital Goods payment flow.

Chapter 4, Adaptive Payments (Simple, Parallel, and Chained Payments)
: Introduces Adaptive Payments and shows you how to implement an Adaptive Payments checkout flow for Tweet Relevance in which funds are sent to multiple recipients.

Chapter 5, Website Payments Pro (Direct Payment)
: Teaches you how to accept credit cards directly from your site using Website Payments Pro's Direct Payment option as a checkout option for Tweet Relevance.

Chapter 6, Instant Payment Notifications (IPNs)
: Demonstrates how to use Instant Payment Notifications (IPNs) to handle custom actions associated with a payment, such as sending a confirmation email to a customer when a purchase is completed.

Appendix A
: Introduces Tweet Relevance, the foundational reference application that's used throughout the book as a baseline project.

Appendix B
: Provides a minimal overview of Mobile Payments Libraries (MPLs). This is also where you can go to get started developing solutions for iOS, Android, and BlackBerry.

It is highly recommended that you read Chapters 1 and 2 before diving into any other chapter, because these initial chapters try to be as thorough as possible in establishing a foundation that future chapters build upon.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*PayPal APIs: Up and Running (2nd Ed.)* by Matthew Russell. Copyright 2012 O'Reilly Media, Inc., 978-1-449-31872-7."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreil.ly/paypal-apis-2e

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1

PayPal API Overview

This chapter provides a very brief overview of PayPal's Name-Value Pair (NVP) APIs, the primary way that you'll interact with PayPal products throughout the remainder of this book. Since using the NVP APIs is as simple as making some HTTP requests and parsing the responses, we'll get a Python project up and running with Google App Engine (GAE) as part of this initial overview. Future chapters all use Python-based GAE projects, so unless you're definitely planning to port the code without running it, you should make the most of this opportunity to learn the basics. After reading this chapter, it is highly recommended that you check out and run the sample code for Tweet Relevance as introduced in Appendix A.

Overview of PayPal API Requests

PayPal's NVP API makes it simple to integrate payments into your application. As the merchant, your web application constructs an NVP string and transmit it via HTTPS (HTTP Secure) to the PayPal authorization server, and PayPal sends back an NVP-formatted response that your web application parses for the information relevant to the payment. Figure 1-1 shows this basic request and response workflow, which is typical of just about any web application.

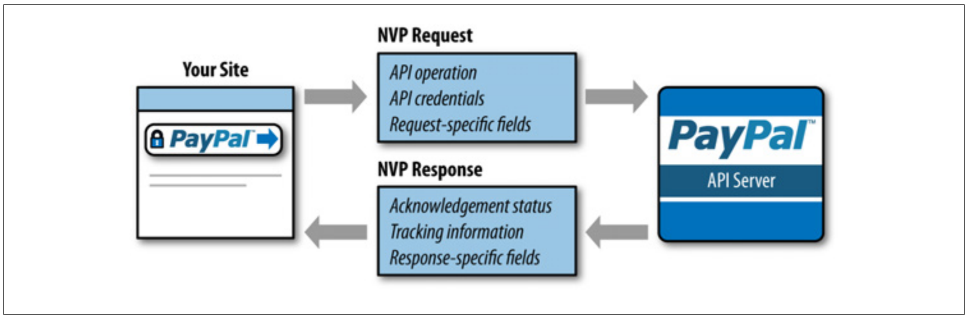


Figure 1-1. Typical NVP request and response

The request identifies:

- The name or method of the API operation to be performed and its version
- PayPal API credentials
- Operation-specific parameters formatted as name/value pairs

Various PayPal products may require additional specific request parameters as indicated by PayPal's official documentation. For example, Adaptive Payments APIs also require an `APP ID` field to be specified.

The PayPal API server executes the operation and returns a response containing:

- Acknowledgment of success or failure (including any warnings returned in case of failure)
- PayPal tracking information specific to the API operation
- Response-specific information required to fulfill the request

Some PayPal products such as Express Checkout require calls to multiple API operations, while others such as Direct Pay (part of Website Payments Pro) only require one call. We'll review Express Checkout in the next chapter, but Figure 1-2 is included to illustrate its typical flow, which should look pretty familiar and intuitive if you've ever used PayPal. Either way, interacting with PayPal products is just a series of API calls that allow you to accomplish a wide variety of tasks. A few examples of the possible transactions PayPal products support include:

- Accepting PayPal as part of a streamlined checkout process
- Charging a credit card
- Capturing previously authorized payments
- Reauthorizing or voiding previous authorizations
- Paying single or multiple recipients
- Issuing full or partial refunds
- Searching transactions histories
- Retrieving details of specific transactions
- Processing payments involving more than one party
- Setting up recurring subscription charges
- Accepting Donations

With a broad understanding of how payment transactions are implemented, let's briefly segue into an overview of GAE and how to implement HTTP requests, the essential skill required to interact with PayPal APIs.

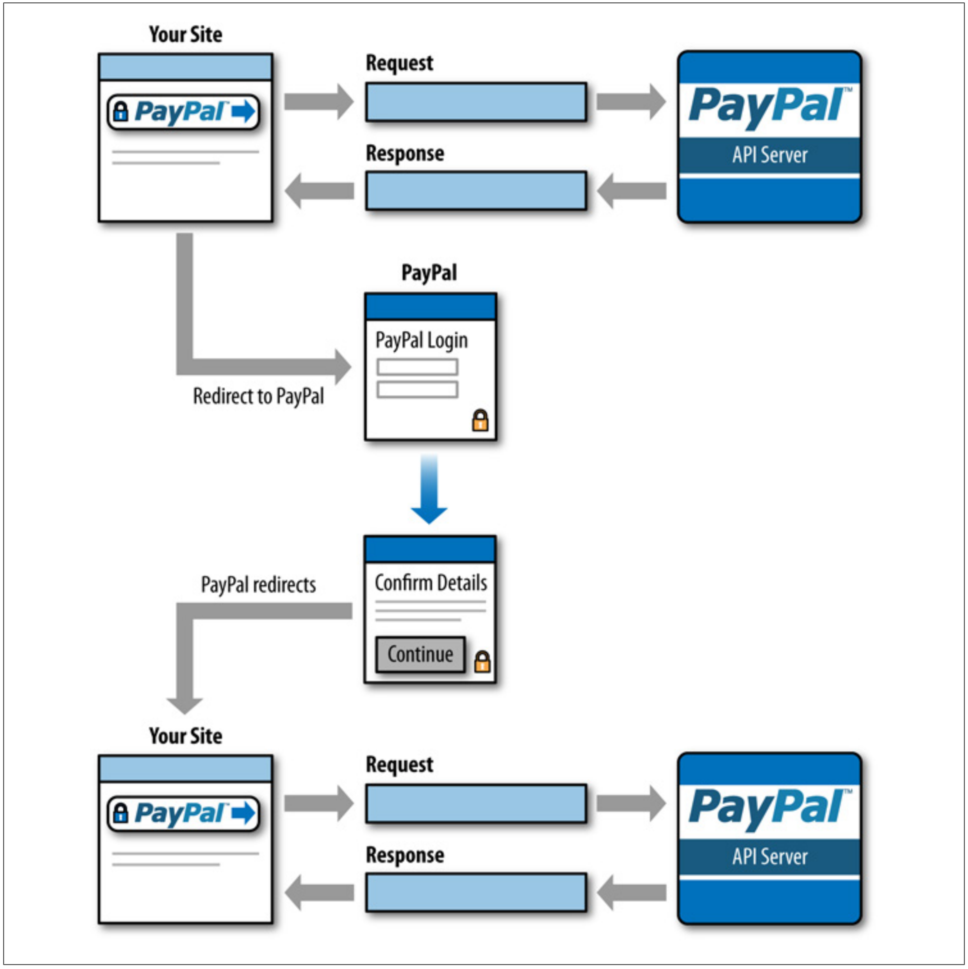


Figure 1-2. A typical Express Checkout in which a merchant site establishes a session with PayPal and then redirects the buyer to PayPal for specification of shipping and payment information. Once the buyer confirms transaction details, PayPal redirects the buyer back to the merchant site where it regains control of the checkout and can issue additional requests to PayPal for final payment processing.

Google App Engine Primer

GAE is a terrific platform, and this book takes advantage of its simplicity and uses it as the standard for communicating how to build web applications that interact with PayPal APIs. It's very easy to get an application up and running locally for test purposes, yet the same applications that you've implemented can be run and scaled out on the very same infrastructure that Google uses for its own applications with virtually no additional work! A vast amount of documentation about GAE is available online, so let's assume that you'll take a little time to familiarize yourself by reviewing the App

Engine Python Overview, which includes a "getting started" guide that walks you through installation of the Python Software Development Kit (SDK).

Building and Deploying Your First App

Assuming you've installed the Python SDK[1] and done little more than use Google App Engine Launcher to create a new sample project, you've essentially already implemented a traditional "Hello, world" program that you can run on your local machine. Launch the program by clicking the Run button, and then click the Browse button to launch and navigate your browser so that you successfully see "Hello world!" to make sure that everything is up and running. Then, take a peek at the contents of *app.yaml* and *main.py*, which are reproduced in Examples 1-1 and 1-2 for convenience. At a high level, the salient points are that the name of the application is `helloworld`; that MainHandler is assigned to the root context of the web application, as indicated by the presence of `('/', MainHandler)` in the list that's supplied to the `WSGIApplication` class constructor; and a `get` method is defined for `MainHandler`, which allows the web application to respond to your browser's GET request when you click the Browse button from the Google App Engine Launcher.

Example 1-1. app.yaml from a stock GAE project

```
application: helloworld
version: 1
runtime: python
api_version: 1

handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  script: main.py
```

Example 1-2. main.py from a stock GAE project

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp import util

class MainHandler(webapp.RequestHandler):
    def get(self):
        self.response.out.write('Hello world!')


def main():
    application = webapp.WSGIApplication([('/', MainHandler)],
                                         debug=True)
```

1. As of this writing, version 1.6.0 is the latest SDK, which supports the Python 2.5 runtime by default. The Python 2.7 runtime is a stable—but still considered experimental—feature.

```
    util.run_wsgi_app(application)


if __name__ == '__main__':
main()
```

At this point, if you naively click the Deploy button to try to deploy the application to the Web, you'll get an unfortunate error in the logging console to the effect of, "You do not have permission to modify this app (app_id=u'helloworld')." In other words, it's telling you that the application id of helloworld that's specified in *app.yaml* is already registered by someone else and that you'll need to try another one. It's unfortunate that the error message doesn't give you a bit more information, because what you really need to do at this point is click the Dashboard button to log into your GAE account and register a web application with a unique identifier, which in turn corresponds to a unique appspot.com subdomain. For example, a GAE web application with an application identifier of `helloworld` would correspond to http://helloworld.appspot.com. You can only register a limited number of application identifiers for free, so it's recommended that you create a generic identifier that you can reuse for multiple test applications. The identifier that'll be used throughout this book is `ppapis2e`, which somewhat corresponds to this book's title, *PayPal APIs: Up and Running (Second Edition)*. You can use whatever identifier you'd like.

You should verify that you can register an application identifier and deploy it to appspot.com before reading further. The steps you should take simply involve:

- Clicking the Dashboard button in the Google App Engine Launcher
- Authenticating into the dashboard with your Google account
- Creating an application in the dashboard
- Changing the top line of your local *app.yaml* file to reflect your web application's name you've chosen
- Clicking the Deploy button in the Google App Engine Launcher
- Navigating your web browser to the corresponding URL on appspot.com that corresponds to the subdomain that you've chosen, i.e., a URL such as http://ppapis2e.appspot.com

There's lots more that could be said about GAE, but we should review at least one more important skill that you'll need before leaving you to the online documentation: implementing HTTP requests. In GAE parlance, this skill is filed under the URL Fetch Python API.

Fetching URLs

One thing that should be mentioned about GAE is that there are some modules from the standard library that are not accessible because of the sensitive nature of running applications in a shared environment. Unfortunately, `urllib`, `urllib2`, and `httplib` are some common modules that you may have used for implementing HTTP requests that are off limits to your GAE application; however, GAE naturally provides ways to make both synchronous and asynchronous requests in a familiar enough manner. Example 1-3 is an updated version of Example 1-2 that makes use of the `urlfetch` function to perform a synchronous HTTP request. (Asynchronous requests are made in a very similar manner except that a callback function defines what should happen once the request completes.) Note the use of the keyword parameter `validate_certificate`, which is employed to ensure that the request is securely completed so as to avoid potential man-in-the-middle attacks. You should be able to deploy the application and verify that it can indeed securely fetch the URL https://paypal.com/ before continuing.

When implementing an online commerce application, always be a bit paranoid and routinely double-check security assumptions.

Example 1-3. An updated main.py that illustrates how to use the urlfetch function to perform a secure HTTP request

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp import util
from google.appengine.api import urlfetch

class MainHandler(webapp.RequestHandler):
    def get(self):
        url = "https://www.paypal.com/"
        result = urlfetch.fetch(
                      url,
                      validate_certificate=True # Avoid man-in-the-middle attacks
                 )
        if result.status_code == 200:
            self.response.out.write('Successfully fetched ' + url)
        else:
           self.response.out.write('Could note fetch %s (%i)' % (url, result.status_code,))


def main():
    application = webapp.WSGIApplication([('/', MainHandler)],
                                         debug=True)
    util.run_wsgi_app(application)

if __name__ == '__main__':
    main()
```

Hopefully, you are now comfortable enough with GAE that you can find your way around and use the online documentation to fill in basic knowledge gaps. Generally speaking, the overall flow for each application is discussed to some degree when sample code is introduced, and inline source code comments are provided wherever helpful, but a basic working knowledge of GAE is assumed moving forward.

Making PayPal API Requests with App Engine

Although PayPal offers a number of SDKs for a number of common programming languages including Java, ASP.NET, Ruby, PHP, and Cold Fusion, the applications in this book use the NVP APIs directly. Visit the SDKs and Downloads section of PayPal's Integration Center for more details about the various SDKs that are available.

In order to make PayPal API requests, you'll first need to register a merchant account and obtain API credentials. Since it wouldn't be a very good idea to implement a commerce application in a live production environment that involves real accounts and real money, PayPal offers a wonderful sandbox environment to use while developing your application. Using the sandbox environment, you can set up faux buyer and seller accounts to fully test the payment flows and track the flow of funds before flipping the switch on your application code and going live. The faux buyer account acts like someone buying a product from a marketplace, and the faux seller account acts like the marketplace that's selling the products. For the most part, switching between the two is as simple as changing the target server URL and the API credentials from sandbox values to production values. The rest of your application will remain unchanged, so it's a fairly painless experience to go live. The remainder of this chapter steps through the process of obtaining API credentials for the sandbox environment, and shows you how to use them to make a PayPal API request.

Obtaining API Credentials for the Sandbox Environment

Developing your application only requires access to the PayPal API sandbox. You can sign up for access to the sandbox environment at *https://developer.paypal.com*. Once your account is established, you can create test accounts for buyers and sellers as well as obtain API credentials. Sandbox accounts and live accounts require different processes to obtain credentials.

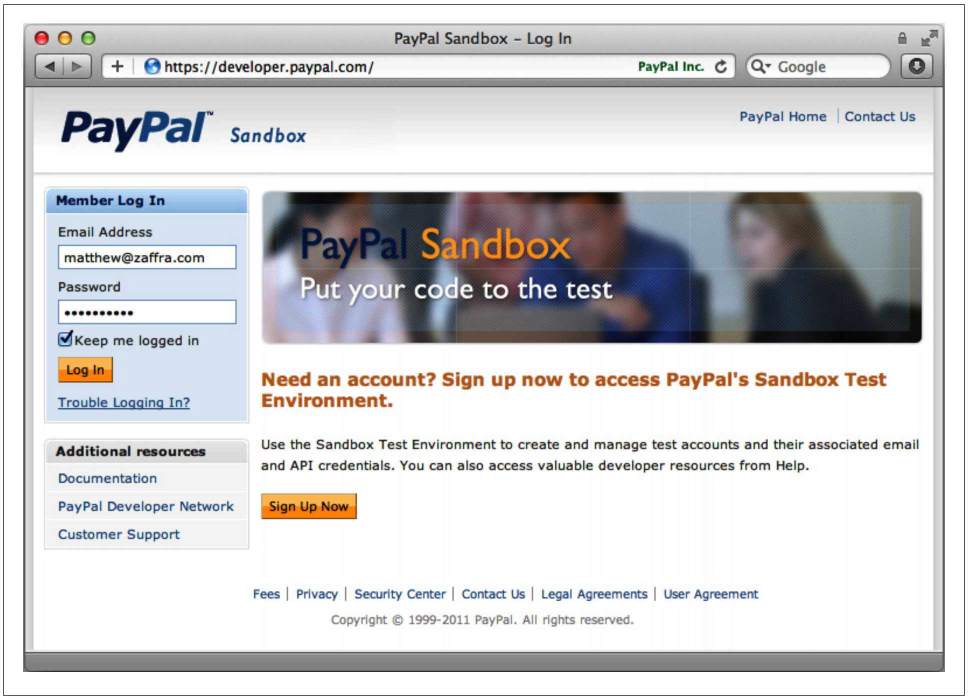
This book doesn't cover some of the nuances of transitioning to a live environment, but you'll essentially just sign up for a merchant account, substitute the production API credentials that come with your merchant account into your web application, and update the API endpoints that your web application uses for making requests. There's also a handy Go Live Checklist that's maintained in PayPal's official documentation that summarizes the key steps.

Use the following steps for a sandbox account:

1. Go to *https://developer.paypal.com* and click Sign Up Now.
2. Enter the requested information and click Agree and Submit.
3. PayPal will send you an email to complete the signup process.
4. After confirming your email address, you can create test accounts and access API credentials by logging into *https://developer.paypal.com/* with the email/password combination you provided during the signup process.
5. Click the Test Accounts link.
6. Click the Create Test Account link.
7. Choose Seller for the account type and select the other appropriate options. Going with the defaults is highly recommended and results in API credentials being created automatically.
8. Click the API credentials link to access your API credentials.
9. Click the Enter Sandbox Test Site button to log in to faux buyer and seller accounts. Note that after the initial login to *https://developer.paypal.com/*, you can log in to individual sandbox accounts by accessing *https://www.sandbox.paypal.com* directly.

You cannot use the same email/password combination to log in to your sandbox account at *https://developer.paypal.com/* that you use to log in to your ordinary PayPal account (unless, of course, you intentionally used the very same email/password combination for both accounts, which is not recommended).

If this is your first encounter with the PayPal Sandbox Test Environment, these steps can seem slightly confusing at first since you end up with so many accounts. The key takeaways are that you can create a developer account and log in to the sandbox environment. From within the sandbox environment, you can create faux buyer and seller accounts, and the creation of a faux seller account provides you with API credentials for the faux merchant account. Note, however, that in order to log in to these faux accounts, you'll need to establish a session by first logging in through *https://www.sandbox.paypal.com* and using the faux account credentials for each individual account; you cannot log in to http://paypal.com with the faux credentials. Figures 1-3 through 1-7 show the overall login flow.


PayPal Sandbox – Log In

https://developer.paypal.com/

PayPal Sandbox

PayPal Home | Contact Us

Member Log In

Email Address: matthew@zaffra.com

Password: ••••••••••

☑ Keep me logged in

Log In

Trouble Logging In?

PayPal Sandbox

Put your code to the test

Need an account? Sign up now to access PayPal's Sandbox Test Environment.

Use the Sandbox Test Environment to create and manage test accounts and their associated email and API credentials. You can also access valuable developer resources from Help.

Sign Up Now

Additional resources

Documentation

PayPal Developer Network

Customer Support

Fees | Privacy | Security Center | Contact Us | Legal Agreements | User Agreement

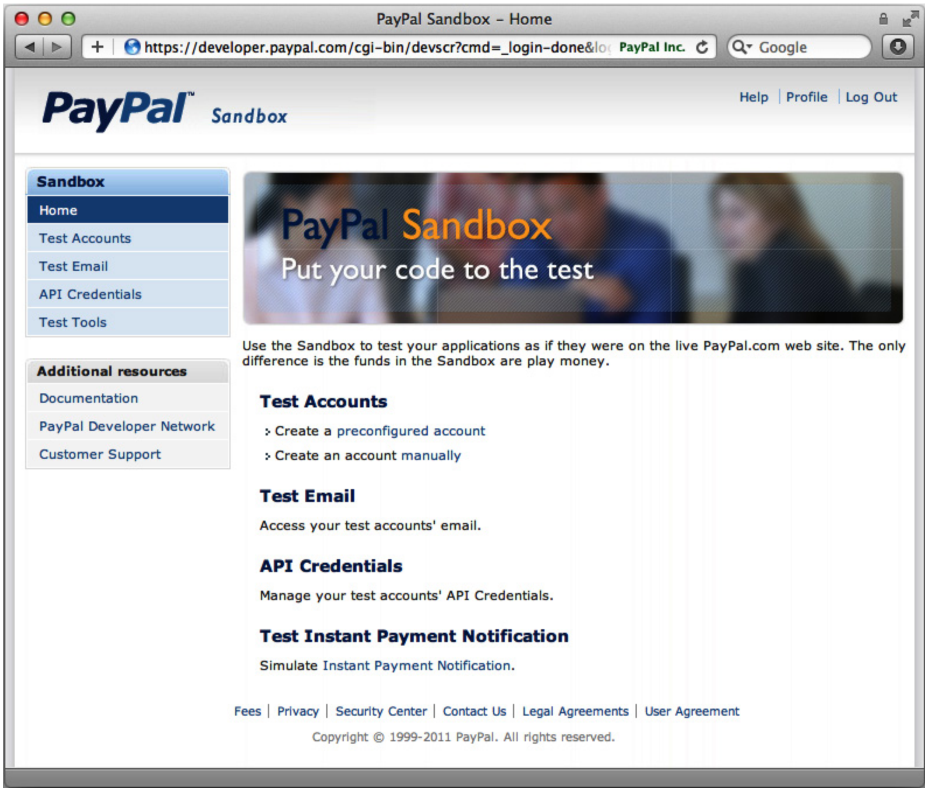
Copyright © 1999-2011 PayPal. All rights reserved.

Figure 1-3. Log in to the sandbox environment with your developer account (which is separate from your ordinary PayPal account).

Once you've successfully worked your way through the developer sandbox fundamentals, you're ready to use your faux merchant account's API credentials to programatically make a request.

Making API Requests with 3-Token Credentials

PayPal offers two methods for authenticating requests: certificates and "3-token credentials," which are comprised of a username, password, and signature. You are already familiar with the concept of securing an account by using a username and a password, but perhaps you're not familiar with the additional signature. Essentially, the signature is an additional password that is intrinsically virtually impossible to guess or crack, and the addition of it to the authentication process results in a scheme known as multi-factor authentication since the signature is an additional factor that is used in addition to the password. By default, faux accounts are set up with 3-token credentials, and we'll use the API Signature method of specifying credentials throughout this book. Figure 1-8 illustrates the 3-token credentials in a developer sandbox account.


PayPal Sandbox – Home

https://developer.paypal.com/cgi-bin/devscr?cmd=_login-done&lo

PayPal Sandbox

Help | Profile | Log Out

Sandbox
- Home
- Test Accounts
- Test Email
- API Credentials
- Test Tools

Additional resources
- Documentation
- PayPal Developer Network
- Customer Support

PayPal Sandbox
Put your code to the test

Use the Sandbox to test your applications as if they were on the live PayPal.com web site. The only difference is the funds in the Sandbox are play money.

Test Accounts
- Create a preconfigured account
- Create an account manually

Test Email
Access your test accounts' email.

API Credentials
Manage your test accounts' API Credentials.

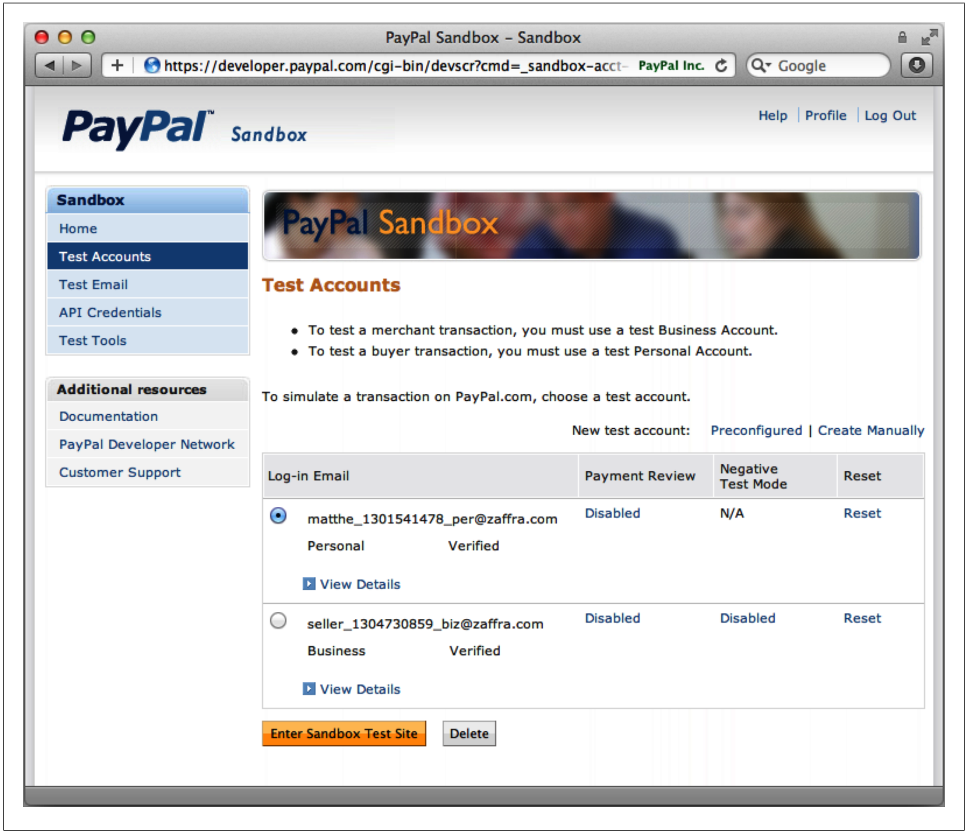
Test Instant Payment Notification
Simulate Instant Payment Notification.

Fees | Privacy | Security Center | Contact Us | Legal Agreements | User Agreement

Copyright © 1999-2011 PayPal. All rights reserved.

Figure 1-4. Once logged into the sandbox environment, you can create and manage test accounts from the Test Accounts menu item.

The remainder of this section works through a few implementation details and culminates with the equivalent of a working "Hello, world!" example to illustrate how to make a PayPal API request and parse the response. Specifically, we'll call the `SetExpressCheckout` API, which is the first step involved in using the Express Checkout product. Express Checkout is featured in depth in the next chapter, so for now, don't worry about the details of what it does. At this point, just think of it as an opaque API operation, and focus on the details of making the request and parsing the response. The key steps your application must accomplish to post to the NVP API include URL encoding, constructing the request in a format the NVP API can interpret, and posting the request via HTTPS to the server. The remainder of this section works through these details.



Test Accounts

- To test a merchant transaction, you must use a test Business Account.
- To test a buyer transaction, you must use a test Personal Account.

To simulate a transaction on PayPal.com, choose a test account.

New test account: Preconfigured | Create Manually

Log-in Email	Payment Review	Negative Test Mode	Reset
● matthe_1301541478_per@zaffra.com Personal Verified View Details	Disabled	N/A	Reset
○ seller_1304730859_biz@zaffra.com Business Verified View Details	Disabled	Disabled	Reset

Figure 1-5. Create faux buyer and seller accounts.

URL encoding and decoding

Both the request to the PayPal server and the response from the server are URL encoded. This method ensures that you can transmit special characters, characters not typically allowed in a URL, and characters that have reserved meanings in a URL. For example:

```
NAME=John Doe&COMPANY=Acme Goods & Services
```

is URL encoded as follows:

```
NAME=John+Doe&Company=Acme+Goods+%26+Services
```

Each application language typically has a specific built-in URL encode method. Refer to the list in Table 1-1 for some specific functions in common programming languages.

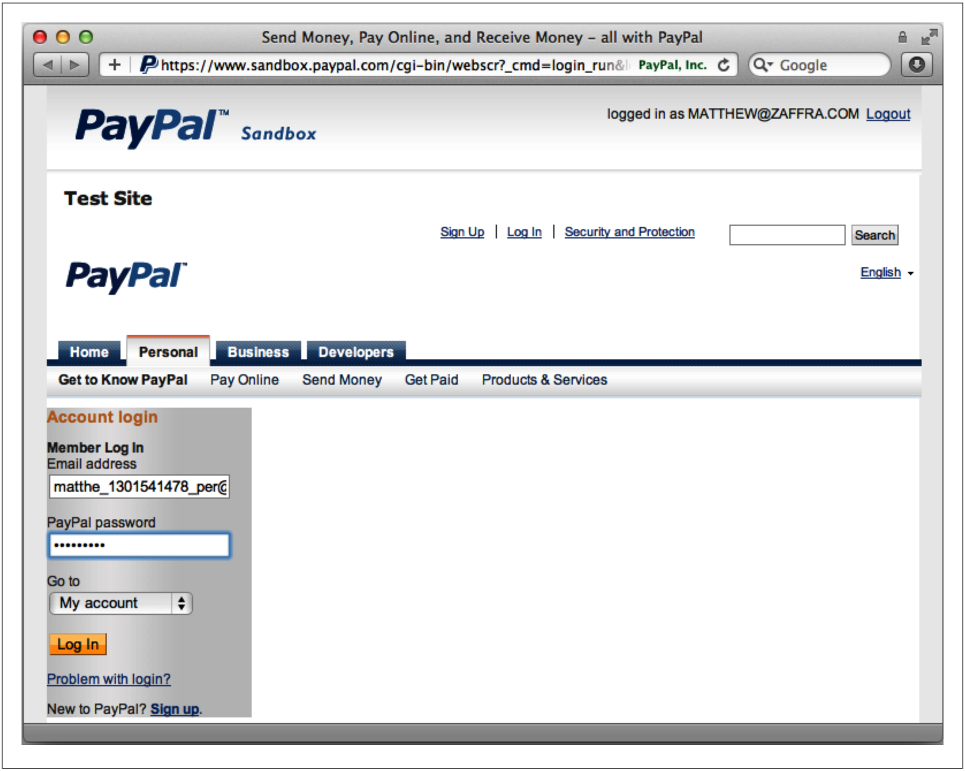
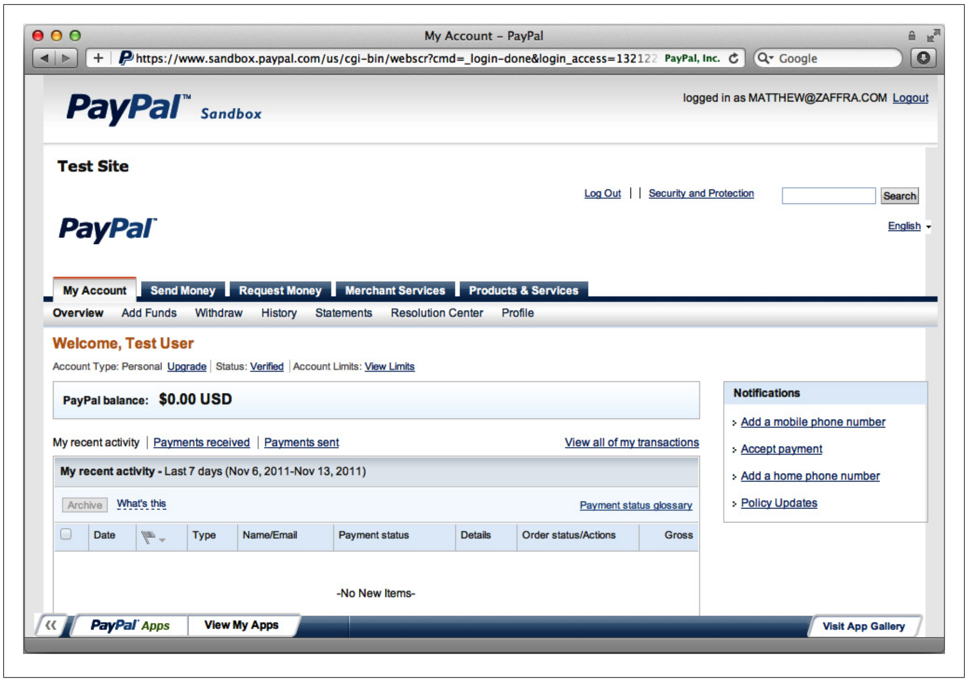


Figure 1-6. Select an account and click Enter Sandbox Test Site to launch a login. (Notice the message in the upper-right corner of the screen that alerts you that you have already established a developer session.)

Table 1-1. URL encoding and decoding methods for common programming languages

Application language	Encoding Function Name	Decoding Function Name
ASP.NET	`System.Web.HttpUtility.UrlEncode`	`System.Web.HttpUtility.UrlDecode`
Classic ASP	`Server.URLEncode`	No built-in function
Java	`java.net.URLEncoder.encode`	`java.net.URLEncoder.decode`
PHP	`urlencode`	`urldecode`
ColdFusion	`URLEncode`	`URLDecode`
Python	`urllib.urlencode`	`urlparse.parse_qs`

Since Python is used as the primary programming language in this book, Example 1-4 illustrates a Python interpreter session showing you how to decode and encode URL query string parameters. The very same logic holds for your GAE app, except that the `parse_qs` function may need be imported from the `cgi` module instead of the `urlparse` module because of changes to Python packages between versions 2.6 and 2.7.



https://www.sandbox.paypal.com/us/cgi-bin/webscr?cmd=_login-done&login_access=132122

logged in as MATTHEW@ZAFFRA.COM Logout

Welcome, Test User

Account Type: Personal Upgrade | Status: Verified | Account Limits: View Limits

PayPal balance: $0.00 USD

My recent activity - Last 7 days (Nov 6, 2011-Nov 13, 2011)

	Date		Type	Name/Email	Payment status	Details	Order status/Actions	Gross

-No New Items-

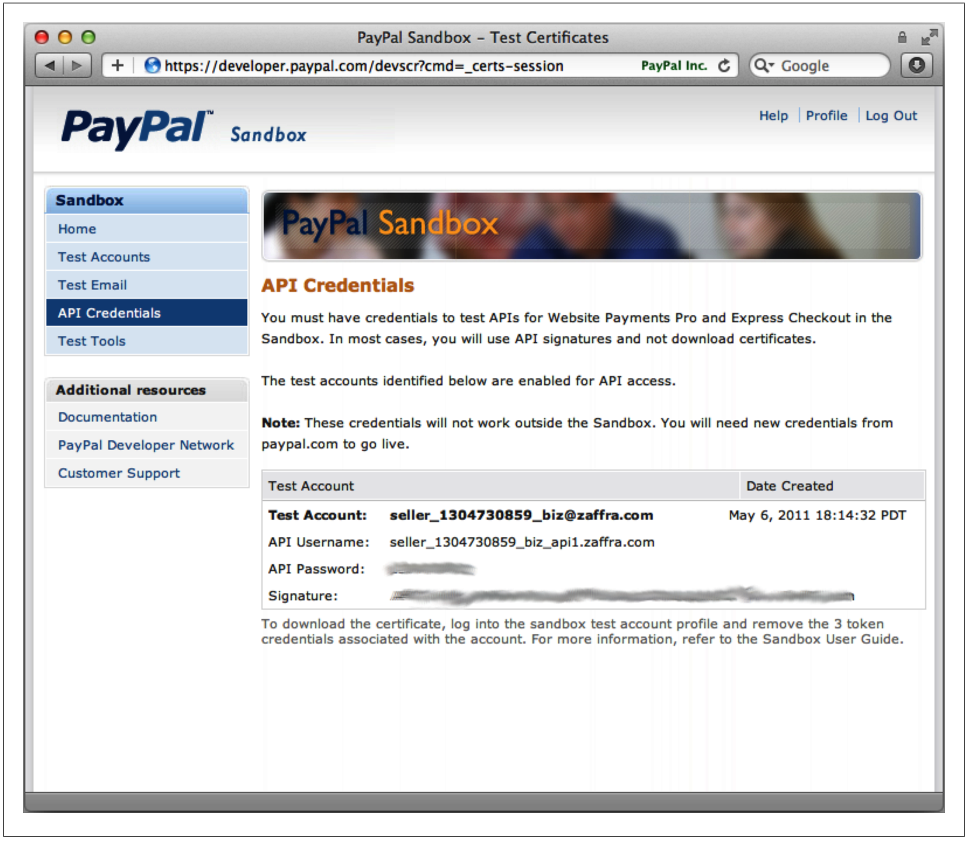
Figure 1-7. Use the faux account credentials to log in to the account—it provides the same view as if it were a real merchant account!

At the time of this writing, Python 2.5 was still the default version to run on GAE, but Python 2.7 seemed imminently releasable and was in an experimental state. Note that as of Python 2.6, `parse_qs` was moved into the `urlparse` module, so attempting to import it from `cgi` will fail if and when GAE defaults to Python version 2.7.

Example 1-4. Encoding and decoding a URL with Python version 2.7

```
>>> import urllib
>>> encoded_qs = urllib.urlencode({'NAME' : 'John Doe', 'COMPANY' : 'Acme Goods &
Services'})
>>> print encoded_qs
'COMPANY=Acme+Goods+%26+Services&NAME=John+Doe'

>>> import urlparse
>>> decoded_qs = urlparse.parse_qs(encoded_qs) # use cgi.parse_qs for Python 2.5 and older
>>> print decoded_qs
{'COMPANY': ['Acme Goods & Services'], 'NAME': ['John Doe']}
```

One very minor nuance to observe is that `parse_qs` returns lists of values for each field in the dictionary result. The reason is that it is legal for URL query string items to be keyed by duplicate field names. For example, `foo=1&foo=2&foo=3` is a valid query string, so `parse_qs` needs to be able to return all of these values back to you uniformly.



API Credentials

You must have credentials to test APIs for Website Payments Pro and Express Checkout in the Sandbox. In most cases, you will use API signatures and not download certificates.

The test accounts identified below are enabled for API access.

Note: These credentials will not work outside the Sandbox. You will need new credentials from paypal.com to go live.

Test Account		Date Created
Test Account:	**seller_1304730859_biz@zaffra.com**	May 6, 2011 18:14:32 PDT
API Username:	seller_1304730859_biz_api1.zaffra.com	
API Password:	[redacted]	
Signature:	[redacted]	

To download the certificate, log into the sandbox test account profile and remove the 3 token credentials associated with the account. For more information, refer to the Sandbox User Guide.

Figure 1-8. PayPal API credentials are available through the developer sandbox environment

Request and response format

Each NVP API request is composed of required and optional parameters and their corresponding values. Parameters are not case-sensitive, but certain values such as the API Password, (`PWD`), are case-sensitive. The required parameters for all NVP API transactions are `USER`, `PWD`, `METHOD`, and `VERSION`. The `METHOD`, or type of transaction you are calling the NVP API to process, has an associated `VERSION`. Together the `METHOD` and `VERSION` define the exact behavior of the API operation you want performed. This will be followed by the information posted from your application, including things such as Item, Quantity, and Cost.

API operations can change between versions, so when you change a version number, be sure to test your application code again before going live.

Each NVP API response is composed of an acknowledgment (or ACK), a timestamp, a CorrelationID unique to the transaction, and a build number stating the API version used to process the transaction. This basic response is then followed by a series of name/value pairs holding the transaction data, which you can parse and handle accordingly in your application. The acknowledgment will be one of the responses outlined in Table 1-2.

PayPal maintains a fairly detailed list of error codes that is very handy to bookmark and consult during debugging situations.

Table 1-2. ACK parameter values

Type of response	Value
Successful response	`Success`, `SuccessWithWarning`
Partially successful response (relevant only for parallel payments; some of the payments were successful and others were not)	`PartialSuccess`
Error response code	`Failure`, `FailureWithWarning`, `Warning`

Making a PayPal Request with GAE

Now that we have established some of the fundamentals, let's make a PayPal API request using GAE. Example 1-5 ties together the common concepts from this chapter and shows you how do it.

Example 1-5. An updated main.py that illustrates how to make a PayPal API request and parse the response

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp import util
from google.appengine.api import urlfetch

import urllib
import cgi

class MainHandler(webapp.RequestHandler):
    def get(self):

        # Sandbox NVP API endpoint

        sandbox_api_url = 'https://api-3t.sandbox.paypal.com/nvp'

        nvp_params = {
            # 3 Token Credentials - Replace XXX with your own values

            'USER' : 'XXX',
            'PWD' : 'XXX',
            'SIGNATURE' : 'XXX',
            # API Version and Operation
            'METHOD' : 'SetExpressCheckout',
            'VERSION' : '82.0',

            # API specifics for SetExpressCheckout
            'PAYMENTREQUEST_0_AMT' : '1.00',
            'RETURNURL' : 'http://ppapis2e.appspot.com/xxx_returnurl_xxx',
            'CANCELURL' : 'http://ppapis2e.appspot.com/xxx_cancelurl_xxx'
        }

        # Make a secure request and pass in nvp_params as a POST payload

        result = urlfetch.fetch(
                    sandbox_api_url,
                    payload = urllib.urlencode(nvp_params),
                    method=urlfetch.POST,
                    validate_certificate=True
                 )

        if result.status_code == 200: # OK

            decoded_url = cgi.parse_qs(result.content)

            for (k,v) in decoded_url.items():
                self.response.out.write('<pre>%s=%s</pre>' % (k,v[0],))
        else:

            self.response.out.write('Could note fetch %s (%i)' %
                    (url, result.status_code,))

def main():
    application = webapp.WSGIApplication([('/', MainHandler)],
                                         debug=True)
    util.run_wsgi_app(application)

if __name__ == '__main__':
    main()
```

Hopefully, the code is pretty self-explanatory. GAE's `urlfetch.fetch` function is used to make a secure request to the sandbox API endpoint, which includes the standard `USER`, `PWD`, and `SIGNATURE` parameters from under the "API Credentials" tab of the sandbox environment, along with an API operation and version as defined by `METHOD` and `VERSION`, respectively. The `SetExpressCheckout` API requires a minimum of a request amount and redirect URL.

Keeping up with updates to PayPal's API isn't quite as simple as one would imagine. The simplest way to keep up with developments is to consult About Previous Versions of the API.

The important thing to take away from this particular example is just that you can substitute in your own 3-token credentials and successfully execute the code. It should run on your local development machine, but also take the opportunity to deploy it as a live application and see it run out on the Web as well.

Example 1-6. Sample results from executing Example 1-5, which calls SetExpressCheckout and displays the results

```
ACK=Success
TIMESTAMP=2011-11-16T15:38:28Z
TOKEN=EC-7JK870639U925801V
VERSION=82.0
BUILD=2256005
CORRELATIONID=f7f4dbb891723
```

Once you've successfully run the code and get back a sample response as shown in Example 1-6, you should be ready to move on to the next chapter, where we'll explore the Express Checkout product in more detail and build a (hopefully) fun and realistic sample app that incorporates it to implement the payment flow. However, before turning to the next chapter, you should definitely check out, review, and run the sample code for Tweet Relevance, the foundational sample project that's used throughout this book and introduced in Appendix A.

Recommended Exercises

- Complete the (official) Python tutorial.
- Review and execute the examples in the Getting Started with Python documentation for GAE.
- Check out, review, and run the sample code for Tweet Relevance, as introduced in Appendix A.
- Bookmark and spend a few minutes browsing PayPal's official documentation.
- Take some time to explore the GAE development console. It provides extensive options for interacting with objects persisted in the data store, an interactive console, and much more. Familiarity with it is essential to efficient GAE software development.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!

CHAPTER 2

Express Checkout (Including Mobile Express Checkout)

Express Checkout is PayPal's premier checkout solution. It allows a customer to check out on your site, log into a PayPal account, and purchase your goods or services. Express Checkout puts PayPal in charge of data security with regard to the customer's billing and credit card information and removes that non-trivial PCI compliance burden and liability from you as the merchant, allowing you to focus on the aspects of your application that differentiate you in the marketplace. In this chapter, we will look at what distinguishes Express Checkout from other generic workflows and take an in-depth look at its integration points. Then, we'll put theory into practice by building a non-trivial (and hopefully, fun) GAE project that hacks on Twitter data and implements Express Checkout for payment processing.

PayPal's official documentation for Express Checkout is available online.

Checkout Process Workflows

Let's start by looking at the process flow of a typical checkout and juxtapose it with an Express Checkout to better understand the underlying value proposition.

Generic Checkout Workflow

Figure 2-1 shows the typical checkout flow a user experiences when buying goods or services online. It includes the following steps:

1. Customer clicks the checkout button on your shopping cart page.
2. Customer enters all shipping information.

3. Customer chooses her payment method and provides all the relevant billing and payment information.
4. Customer reviews order and pays.
5. Customer receives her order confirmation.

As you can see, this typical checkout method requires the customer to provide a lot of information at the time of purchase. Numerous studies have shown that a cumbersome checkout process is a sure way to lose customers. As you're about to see, Express Checkout can be a real time saver for your customers and translate into higher conversions.

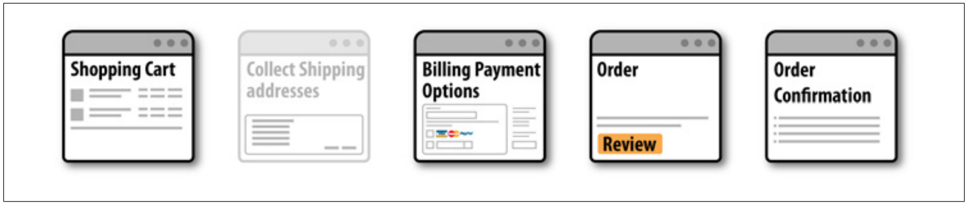


Figure 2-1. Generic checkout workflow

Express Checkout Workflow

Figure 2-2 shows an Express Checkout. Take special note that the process is considerably more streamlined. There's no need to enter shipping information or specifics for payment information. For the typical case, there's literally just a few clicks and the order is processed.

1. Customer chooses Express Checkout by clicking the "Check out with PayPal" button on your site.
2. Customer logs into PayPal.
3. Customer reviews the transaction on PayPal.
4. Customer confirms the order and pays from your site.
5. Customer receives an order confirmation.

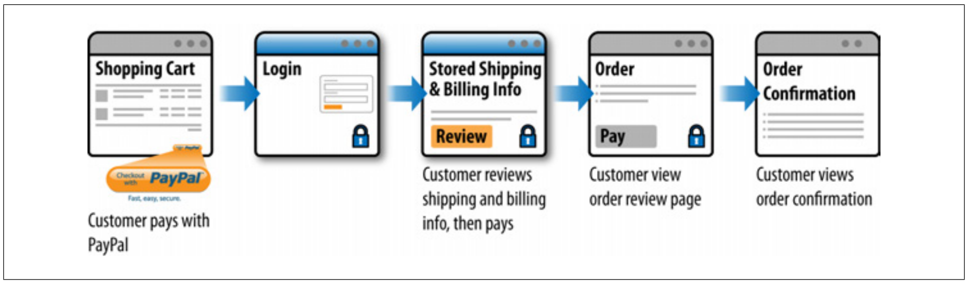

Shopping Cart
Checkout with PayPal
Fast, easy, secure.
Customer pays with PayPal
Login
Stored Shipping & Billing Info
Review
Customer reviews shipping and billing info, then pays
Order
Pay
Customer view order review page
Order Confirmation
Customer views order confirmation

Figure 2-2. PayPal Express Checkout workflow

With Express Checkout, the customer does not need to enter his billing and shipping information each time. Consequently, customers can make purchases and move on to other tasks much more quickly. Table 2-1 outlines the steps required to complete a payment during a generic checkout and Express Checkout. As you can see, Express Checkout saves both time and processing steps.

Table 2-1. Generic checkout versus Express Checkout

Checkout step	Generic checkout	Express Checkout
Select the checkout button	✓	✓
Enter shipping info	✓	-
Select payment method	✓	-
Enter payment information	✓	-
Review order	✓	✓
Confirm order	✓	✓

Express Checkout Flow

To fully implement Express Checkout, you must allow your customers two entry points into the Express Checkout payment process. Figure 2-3 outlines the complete checkout flow for Express Checkout. In short, customers can enter into the Express Checkout flow at either the Shopping Cart Checkout entry point (dotted arrow) or the Payment Methods entry point (solid arrow). Although it might seem a bit curious that there are two entry points to initiate the Express Checkout flow, the basic premise is that the Shopping Cart Checkout entry point gives customers familiar with PayPal an immediate opportunity to perform the quickest checkout possible, while the Checkout entry point essentially provides PayPal as an option alongside other possible payment options. Depending on your perspective, the existence of two entry points could be seen as a little extra nudge for customers to use PayPal for checkout, although from a functional standpoint, having the two entry points does make sense: one provides the speediest checkout possible, and the other presents PayPal in the same context as any other payment method. At any rate, including both methods in your checkout routines is recommended and easy to implement.

Figure 2-4 outlines the Checkout Entry Point, which requires the following steps:

1. Customer clicks the "Check out with PayPal" button.
2. Customer logs into PayPal.
3. Customer confirms shipping and billing information on PayPal's site.
4. Customer is returned to your application for final review and clicks the Purchase button.
5. Customer is returned to a confirmation screen related to the purchase.

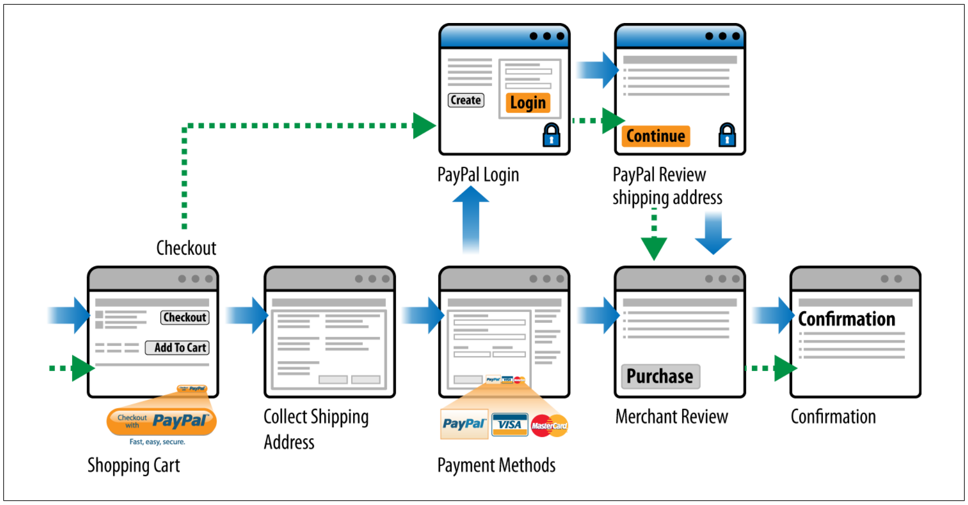


Figure 2-3. Complete Express Checkout flow

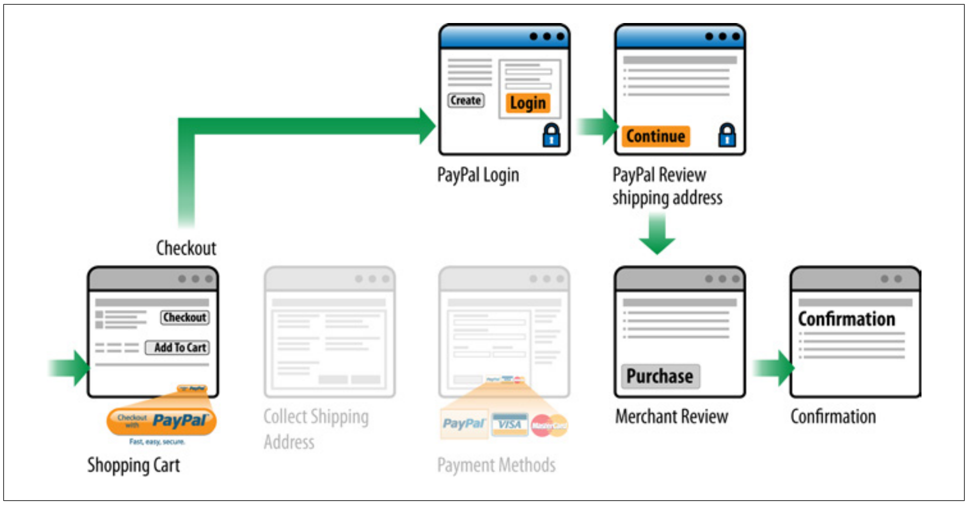


Figure 2-4. Checkout Entry Point

Figure 2-5 outlines the Payment Method Entry Point, which requires the following steps:

1. Customer clicks the Checkout button on your application.
2. Customer inputs shipping information into your application.
3. Customer chooses PayPal from the list of payment methods.
4. Customer logs into PayPal.
5. Customer reviews payment information on PayPal's site.

6. Customer is returned to your application for final review and clicks the Purchase button.
7. Customer is returned to a confirmation screen related to the purchase.

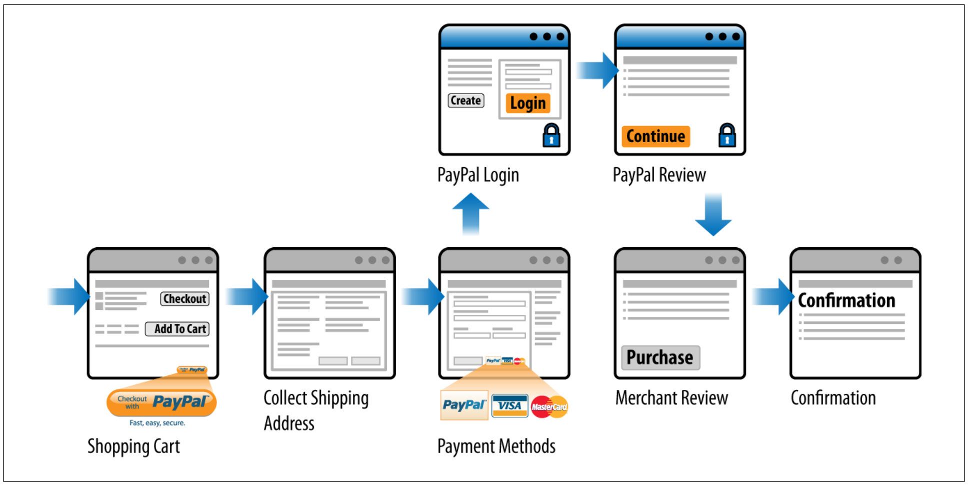


Figure 2-5. Payment Method Entry Point

With a conceptual understanding of Express Checkout and its two entry points now in place, let's take a look at the implementation details involved in an integration.

PayPal Express Checkout API Operations

The PayPal NVP API provides three primary methods related to Express Checkout. These operations initialize the transaction, obtain information about the buyer, and complete the transaction. Table 2-2 outlines these methods and some of the most fundamental parameters for each of them.

Table 2-2. Express Checkout API operations

API operation	Description
`SetExpressCheckout`	Sets up the Express Checkout transaction. You can specify information to customize the look and feel of the PayPal site and the information it displays. At a minimum, you must specify the following information: • URL to the page on your website to which PayPal redirects after the buyer logs into PayPal and approves the payment successfully. • URL to the page on your website to which PayPal redirects if the buyer cancels the transaction. • Total amount of the order or your best estimate of the total. (Although exact shipping or handling amounts may not yet be known, this value should be estimated as accurately as possible.)

API operation	Description
`GetExpressCheckoutDetails`	Obtains information about the buyer from PayPal, including shipping information.
`DoExpressCheckoutPayment`	Completes the Express Checkout transaction, including the actual total amount of the order.

Figure 2-6 provides an overview of the Express Checkout user experience from a software developer's perspective. Before digging in deeper, it may be helpful to think through the following cursory explanation of these Express Checkout integration points. For the purposes of this checkout flow, let's assume that a buyer has just initiated a checkout action on your site by clicking a "Pay with PayPal" button (the Checkout Entry Point) and will choose to complete the checkout process.

Set Express Checkout

- Behind the scenes, you invoke the `SetExpressCheckout` API and pass along details about the order, such as how much it costs and where to redirect the user when the transaction is completed or cancelled.
- You redirect the buyer to paypal.com by constructing a URL that includes a token parameter returned in the response from `SetExpressCheckout` so that PayPal can identify the buyer and details of the purchase, such as its amount.
- The buyer confirms shipping and payment information for the purchase directly on paypal.com, and PayPal redirects the buyer back to your website along with `token` and `PayerID` parameters.

Get Express Checkout Details

- Behind the scenes, you invoke the `GetExpressCheckoutDetails` API operation using the token parameter to request details about the purchase, such as where to ship it, an email address to send an order confirmation, etc.
- The buyer performs a final review of the order, including pertinent details you may have just fetched from PayPal, and finalizes the purchase by clicking a Complete Purchase button.

Do Express Checkout Payment

- Behind the scenes, you invoke the `DoExpressCheckoutPayment` API using the `token` and `PayerID` parameters to identify the user and securely complete the purchase.
- You display an order confirmation and optionally take additional actions, such as sending the buyer an email confirmation or shipping a physical item.

The remainder of this section takes a closer look at each of these API operations.

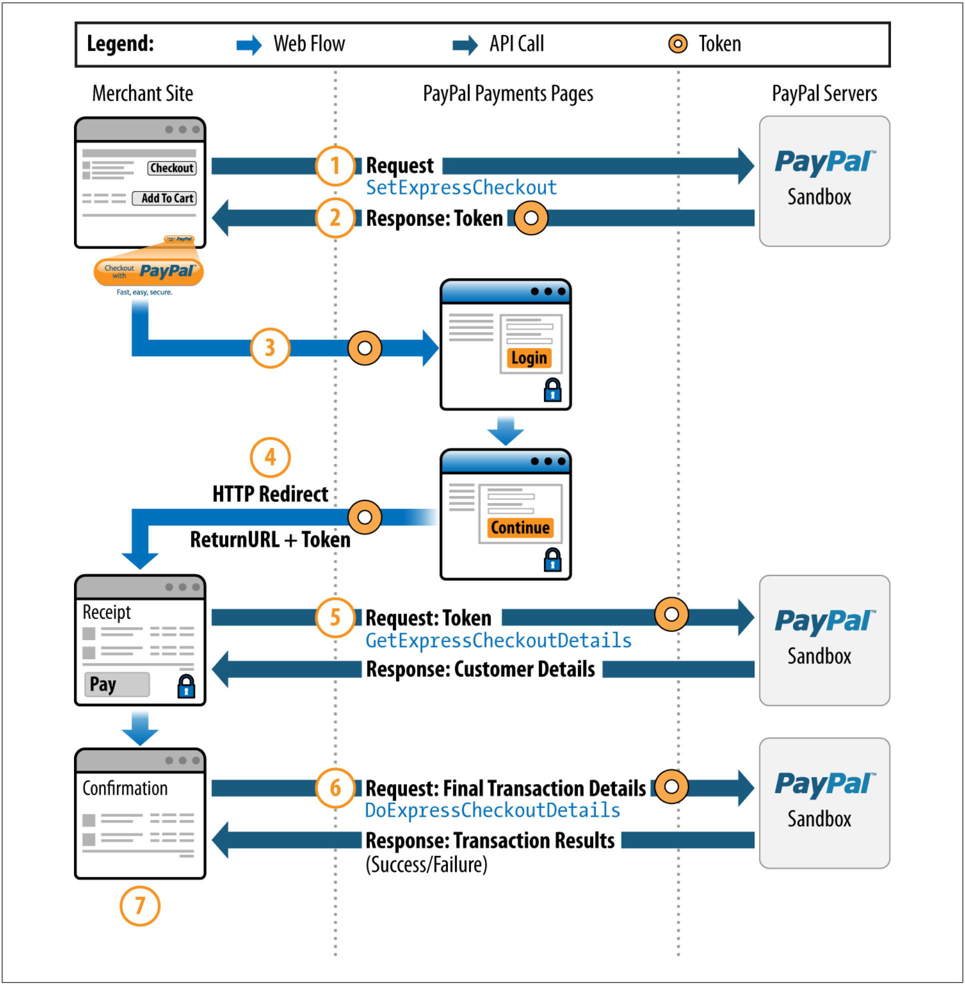


Figure 2-6. Express Checkout Integration Points

A few special caveats such as additional required parameters for requests do apply for Express Checkout integrations involving digital goods purchases and some Adaptive Payments scenarios. This chapter overlooks these caveats and focuses on a typical integration. See Chapters 3 and 4 for specifics on Digital Goods purchases and using Adaptive Payments, respectively.

SetExpressCheckout

`SetExpressCheckout` initializes the Express Checkout session and is the same operation you executed in "Making a PayPal Request with GAE" on page 15 to make your first

API request to PayPal. It allows you to pass variables that format how the PayPal pages look and specify where to redirect the buyer's browser based upon success of the payment transaction. Table 2-3 outlines the fields required for `SetExpressCheckout` requests, and Table 2-4 outlines the fields you can expect back in a response for `SetExpressCheckout`.

See SetExpressCheckout API Operation for full details on a litany of other optional fields that can be passed in for `SetExpressCheckout`.

Table 2-3. SetExpressCheckout request fields

Field	Description
`METHOD`	Must be `SetExpressCheckout`
`RETURNURL`	URL to which the customer's browser is returned after choosing to pay with PayPal. PayPal recommends that the value be the final review page on which the customer confirms the order and payment or billing agreement. Limitation: Up to 2,048 characters.
`CANCELURL`	URL to which the customer is returned if he does not approve the use of PayPal to pay you. PayPal recommends that the value be the original page on which the customer chose to pay with PayPal or establish a billing agreement. Limitation: Up to 2,048 characters.
`PAYMENTREQUEST_`*n*`_AMT`	The total cost of the transaction to the customer. If shipping and tax charges are known, include them in this value; if not, this value should be the current subtotal of the order. If the transaction includes one or more one-time purchases, this field must equal the sum of the purchases. Set this field to 0 if the transaction does not include a one-time purchase—for example, when you set up a billing agreement for a recurring payment that is not charged immediately. Purchase-specific fields will be ignored. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).

If you have done any previous work with Express Checkout, you may want to take note that `AMT` has been deprecated in favor of `PAYMENTREQUEST_`*n*`_AMT` to accommodate tallying up multiple items in a checkout. Simply use `PAYMENTREQUEST_0_AMT` if dealing with only one item. Additionally, `PAYMENTACTION` is deprecated in favor of `PAYMENTREQUEST_`*n*`_PAYMENTACTION` and is no longer a required field, and it now defaults to `Sale`.

Table 2-4. SetExpressCheckout response fields

Field	Description
`TOKEN`	A time-stamped token that acts as a session identifier that is used in subsequent API requests to tell PayPal that you are processing this payment with Express Checkout.

Field	Description
	The token expires after three hours. If you set the token in the SetExpressCheckout request, the value of the token in the response is identical to the value in the request.

GetExpressCheckoutDetails

GetExpressCheckoutDetails obtains information about an Express Checkout transaction. The response essentially echoes back the information and values enabled in SetExpressCheckout, although it does return a few other important fields such as PAYERID, and it is possible that the response may contain additional fields such as NOTE. For example, the response may contain a NOTE field that the buyer may have entered in if the ALLOWNOTE field was set in SetExpressCheckout. Table 2-5 describes the required GetExpressCheckoutDetails fields. As a best practice, it is recommended that you invoke GetExpressCheckoutDetails as part of an Express Checkout integration, but it is not technically required that you do so.

See GetExpressCheckoutDetails API Operation for full details on this API.

Table 2-5. GetExpressCheckoutDetails request fields

Field	Description
METHOD	Must be GetExpressCheckoutDetails
TOKEN	The same time-stamped token as returned by the SetExpressCheckout response

DoExpressCheckoutPayment

DoExpressCheckoutPayment completes the Express Checkout transaction and returns the payment response. In the case of a billing agreement that you specified in the SetExpressCheckout API call, it is officially created when you call the DoExpressCheckoutPayment API operation. For a minimal Express Checkout integration, you'd only need to properly invoke DoExpressCheckoutPayment after first setting up the transaction with SetExpressCheckout and handling the PayPal redirects. Table 2-6 lists some common DoExpressCheckoutPayment request fields. The number of fields returned from DoExpressCheckoutPayment is rather extensive. Although you may not necessarily need to use all of these values, Table 2-7 describes some of them to give you an idea of the breadth of information that's available to you after a checkout has been completed.

Table 2-6. DoExpressCheckoutPayment request fields

Field	Description
METHOD	Must be DoExpressCheckoutPayment.
TOKEN	A time-stamped token, the value of which was returned by the SetExpressCheckout response and passed on to the GetExpressCheckoutDetails request.

Field	Description
PAYERID	Unique PayPal customer account identification number. This value is obtained by parsing the query string immediately after PayPal redirects the customer back to your site or by parsing the results returned from GetExpressCheckoutDetails.
PAYMENTREQUEST_*n*_AMT	The total cost of the transaction to the customer. If shipping and tax charges are known, include them in this value; if not, this value should be the current subtotal of the order. If the transaction includes one or more one-time purchases, this field must equal the sum of the purchases. Set this field to 0 if the transaction does not include a one-time purchase, for example, when you set up a billing agreement for a recurring payment that is not charged immediately. Purchase-specific fields will be ignored. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
PAYMENTREQUEST_*n*_PAYMENTACTION	How you want to obtain your payment. • Sale indicates that this is a final sale for which you are requesting payment (this is the default). • Authorization indicates that this payment is a basic authorization subject to settlement with PayPal Authorization and Capture. • Order indicates that this payment is an order authorization subject to settlement with PayPal Authorization and Capture. If the transaction does not include a one-time purchase, this field is ignored. You cannot set this value to Sale in SetExpressCheckout request and then change this value to Authorization or Order on the final API DoExpressCheckoutPayment request. If the value is set to Authorization or Order in SetExpressCheckout, the value may be set to Sale or the same value (either Authorization or Order) in DoExpressCheckoutPayment.

Table 2-7. DoExpressCheckoutPayment response fields

Field	Description
TOKEN	A time-stamped token, the value of which was returned by the SetExpressCheckout response.
PAYMENTTYPE	Information about the payment.
SUCCESSPAGEREDIRECTREQUESTED	Flag that indicates whether you need to redirect the customer to back to PayPal after completing the transaction.

Field	Description
PAYMENTINFO_*n*_TRANSACTIONID	Unique transaction ID of the payment. If the PaymentAction of the request was Authorization or Order, this value is your AuthorizationID for use with the Authorization and Capture APIs.
PAYMENTINFO_*n*_TRANSACTIONTYPE	The type of transaction. Valid values are cart and express-checkout.
PAYMENTINFO_*n*_PAYMENTTYPE	Indicates whether the payment is instant or delayed. Valid values are none, echeck, and instant.
PAYMENTINFO_*n*_ORDERTIME	The time/date stamp of the payment.
PAYMENTINFO_*n*_AMT	The final amount charged, including any shipping and taxes from your Merchant Profile.
PAYMENTINFO_*n*_FEEAMT	PayPal fee amount charged for the transaction.
PAYMENTINFO_*n*_TAXAMT	Tax charged on the transaction.
PAYMENTINFO_*n*_EXCHANGERATE	Exchange rate if a currency conversion occurred. Relevant only if you are billing in the customer's nonprimary currency. If the customer chooses to pay with a currency other than the primary currency, the conversion occurs in the customer's account.
PAYMENTINFO_*n*_PAYMENTSTATUS	The status of the payment, which will be one of the following: • None: No status. • Canceled-Reversal: A reversal has been canceled, for example, when you win a dispute and the funds for the reversal are returned to you. • Completed: The payment has been completed and the funds have transferred successfully to your account. • Denied: You denied the payment. This will occur only if the payment was previously pending for reasons described in the PendingReason field. • Expired: The authorization period for the payment has expired. • Failed: The payment failed. This occurs only when the payment was made from your customer's bank draft account. • In-Progress: Transaction has not terminated, most likely due to an authorization awaiting completion. • Partially-Refunded: Payment has been partially refunded. • Pending: Payment is still pending for reasons described in the PendingReason field. • Refunded: You refunded the payment. • Reversed: Payment was reversed due to a charge back or other reversal. The funds have been removed from your account balance and returned to the buyer. The reason will be described in the ReasonCode field. • Processed: Payment has been accepted. • Voided: Authorization for the transaction has been voided.

Field	Description
PAYMENTINFO_*n*_ PROTECTION ELIGIBILITY	The type of seller protection in force for the transaction, which is one of the following values: • Eligible: Seller is protected by PayPal's Seller protection policy for Unauthorized Payments and Item Not Received. • PartiallyEligible: Seller is protected by PayPal's Seller Protection Policy for Item Not Received. • Ineligible: Seller is not protected under the Seller Protection Policy.
PAYMENTREQUEST_*n*_ PAYMENTREQUESTID	The unique identifier of the specific payment request. The value should match the one passed in the DoExpressCheckout request.
L_PAYMENTINFO_*n*_FMFfilterIDn	Filter ID, including the filter type (PENDING, REPORT, or DENY), the filter ID, and the entry number, *n*, starting from 0. Filter ID is one of the following values [AVS stands for Address Verification System]: • 1 = AVS No Match • 2 = AVS Partial Match • 3 = AVS Unavailable/Unsupported • 4 = Card Security Code (CSC) Mismatch • 5 = Maximum Transaction Amount • 6 = Unconfirmed Address • 7 = Country Monitor • 8 = Large Order Number • 9 = Billing/Shipping Address Mismatch • 10 = Risky Zip Code • 11 = Suspected Freight Forwarder Check • 12 = Total Purchase Price Minimum • 13 = IP Address Velocity • 14 = Risky Email Address Domain Check • 15 = Risky Bank Identification Number (BIN) Check • 16 = Risky IP Address Range • 17 = PayPal Fraud Model
L_PAYMENTINFO_*n*_FMFfilterNAME*n*	Filter name, including the filter type (PENDING, REPORT, or DENY), the filter NAME, and the entry number, *n*, starting from 0.
PAYMENTREQUEST_*n*_SHORTMESSAGE	Payment error short message.
PAYMEMNTREQUEST_*n*_LONGMESSAGE	Payment error long message.
PAYMENTREQUEST_*n*_ERRORCODE	Payment error code.
PAYMENTREQUEST_*n*_SEVERITYCODE	Payment error severity code.
PAYMENTREQUEST_*n*_ACK	Application-specific error values indicating more about the error condition.

Field	Description
SHIPPINGCALCULATIONMODE	Describes how the options that were presented to the user were determined, and is one of the following values: • API - Callback • API - Flatrate
INSURANCEOPTIONSELECTED	The Yes/No option that you chose for insurance.
SHIPPINGOPTIONISDEFAULT	Is true if the buyer chose the default shipping option. Value will be either TRUE or FALSE.
SHIPPINGOPTIONAMOUNT	The shipping amount that was chosen by the buyer. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
SHIPPINGOPTIONNAME	This is true if the buyer chose the default shipping option.
PAYMENTREQUEST_ *n* _ SELLER PAYPAL ACCOUNTID	Unique identifier for the merchant. For parallel payments, this field contains either the Payer ID or the email address of the merchant.

See DoExpressCheckoutPayment API Operation for more details on this API, but do note that it's well worth the time to peruse the full details of this lengthy table, even though it might seem a bit pedantic to at first—a misinformed understanding of this table can result in faulty logic in your implementation that can really hurt your bottom line. The sidebar "Instant Payments versus eChecks" explains one possible scenario involving a commonly overlooked payment type involving what's known in PayPal lingo as an eCheck.

Instant Payments versus eChecks

Did you notice that in Table 2-7, the PAYMENTINFO_ *n* _PAYMENTTYPE field can return several different values, including none, echeck, and instant? Or that PAYMENTINFO_ *n* _PAYMENTSTATUS can have over a dozen possible values? Knowing the difference between an eCheck and an instant payment is a very important implementation detail, as is inspecting the result of the PAYMENTINFO_ *n* _PAYMENTSTATUS value. In short, instant payments are transferred to your merchant accounts immediately when a buyer uses funds from a PayPal account or ends up using a credit/debit card as a backup funding source that's eligible for an instant payment by means of having available credit. eChecks are used for payment when a buyer doesn't have available funds from a PayPal account and doesn't have a credit/debit card that's eligible for instant payment, which usually means that PayPal debits the funds from a linked bank account. Like an ordinary paper check, it can unfortunately take a few days for the whole process to resolve and for the check to finally clear.

eChecks introduce some additional complexity in the implementation of a payment solution because as the seller, you are essentially in limbo if you receive an eCheck with a status of "pending." Whereas an instant payment is guaranteed money, an eCheck, just like a paper check, may "bounce" and ultimately resolve to a status such as

`Denied`, which means that you won't ever be receiving the funds. Thus, it's imperative that you be prepared to handle eChecks; however, the ability to properly handle eChecks requires an understanding of Instant Payment Notifications (IPNs), a message delivery service that PayPal uses to notify you when an eCheck's status is finalized. Since IPNs aren't introduced until Chapter 6, the sample code for Chapters 2 through 5 is designed to process only instant payments for simplicity.

If you'd like to test an application in the sandbox using an eCheck, create a sample account with a lower bank account balance than the purchase price and do not add a credit card to the account. For non-digital goods purchases, you'll observe that PayPal gives you the eCheck option. For digital goods purchases, however, PayPal prompts you to add a credit card since digital goods purchases require instant payments by virtue of being delivered (almost) instantly in most cases.

Implementing a Checkout Experience for Tweet Relevance

Let's take the newfound knowledge from this chapter and use it to implement an Express Checkout for Tweet Relevance, the sample project described in Appendix A that we'll be using throughout this book. Recall that Tweet Relevance is essentially a mechanism to rank the tweets in a Twitter user's home timeline by relevance instead of the de facto chronological view that's typical of most Twitter user interfaces. Since so many Twitter users are plagued by information (tweet) overload, it seems reasonable to think that they would be willing to pay a nominal fee for access to a service that curates tweets that appear in their home timelines. The remainder of this section involves the selection of a payment model and the details involved in integrating an Express Checkout.

Selecting a Payment Model

The primary detail that we'll need to settle upon before implementing a payment flow for Tweet Relevance is the payment model. In other words, we must determine what the specific product is that a buyer is purchasing in order to access Tweet Relevance. Although there are perhaps other options that could make sense, a few primary options come to mind:

Fixed-rate access
: In fixed-rate access (pay per access) model, a user might pay a small fee in order to access the service for a specified duration. For example, a special "trial access" product to the service might cost $0.99 and provide unlimited access for 24 hours, unlimited access for 30 days might cost $9.99, and unlimited access for 365 days access might cost $99.99.

Recurring subscription
: A subscription service might provide unlimited access to the application for a longer-term duration such as a month or year. For example, a subscription might cost $8.99 for 30 days of unlimited access or $89.99 for 365 days of unlimited

access. It is fairly common that subscriptions automatically renew once they expire and sometimes offer a discount in comparison to a fixed-rate access model as an incentive.

Virtual currency
: A virtual currency such as “login tokens” might allow users to log in and access the application for a limited duration. For example, a bundle of 50 tokens might cost $4.99, and users would expend a single token each time that they login to access the application.

To get up and running with Express Checkout in this chapter, let’s opt to implement the fixed-rate access payment model for Tweet Relevance.

Injecting an Express Checkout Entry Point into Tweet Relevance

A minimal integration with Express Checkout is pretty straightforward. Recalling from Appendix A that users must access `/app` to access the application, this particular API seems to be a reasonable place to inject a Checkout Entry Point to kick off the payment flow. Conceptually, all that needs to happen is a redirect to a special page that displays a “Checkout with PayPal” button and alerts the user that they must pay to access the service. From there, clicking the button kicks off an Express Checkout, which ultimately returns the user to the application once a payment is successfully processed. Having a basic understanding of how Tweet Relevance is designed from reviewing Appendix A and its source code, let’s try to come up with a detailed design for injecting a Checkout Entry Point:

Add `User` *and* `Product` *classes*
: In order to implement a payment model, it’s necessary to create abstractions for user accounts and products. The `Product` class should feature only a single product that is 30 days of access to the application for $9.99. The `User` class should be a minimalist account abstraction that keeps enough state for determining whether or not an account is in good standing based on when access was purchased and how much access was purchased.

Add a `PaymentHandler` *module to the* `handlers` *package*
: Tweet Relevance has an `AppHandler` module that encapsulates core application logic and public APIs, so let’s add a separate `PaymentHandler` that provides the same kind of encapsulation for payment-related logic and APIs.

Add a `paypal` *package*
: We could further refine the logic in `PaymentHandler` by separating the core logic associated with making API requests for Express Checkout from the more general GAE web app logic associated with the `PaymentHandler`.

Modify `AppHandler` *to limit access based on* `Users`
: After users authenticate with their Twitter account, check to see whether or not they have an account in good standing. If they don’t have an account in good

standing, redirect them to a Checkout Entry Point so that they can create or reconcile an account to gain access.

Add a template for the Checkout Entry Point into `AppHandler`
: We'll add a separate template that can serve as a Checkout Entry Point and have `AppHandler` serve it up as needed to kick off an Express Checkout.

The sample code for Tweet Relevance is essentially stateless. Aside from memcache being used to implement a minimalist session so that the application can serve data to a rich Ajax client, the application does not store any persistent state. The addition of an account abstraction such as `User`, however, requires persisting state in a reliable and fault-tolerant datastore since customers rightly expect to receive access to the application once they have paid for it. Fortunately, GAE provides just the kind of datastore we need, and the basics of integrating with it are fairly intuitive. We won't be doing anything very advanced, but it is nonetheless recommended that you bookmark and review The Python Datastore API if you're not familiar with it.

Examples 2-1 and 2-2 introduce the `User` and `Product` modules, which are added to the top level of the project in *User.py* and *Product.py* files. `Product` is just a stub method that returns static product information that could otherwise be managed in a more sophisticated way, but is sufficient for our purposes. `User` stores the minimal information necessary to limit account access based on the payment model selected: an account identifier, when access began, and how long access should last. Whether or not an account is in good standing can be computed by subtracting the `User`'s `access_start` field from the current value returned by `datetime.datetime.now()` and inspecting the number of days in the resulting `datetime.timedelta` value.

The substantive updates to `AppHandler` are shown in Example 2-3. `AppHandler` directly calls `accountIsCurrent` as a helper method since different payment models would have different criteria for determining whether or not an account is current. The `creditUserAccount` method is exposed as a static method so that it may trivially be invoked by `PaymentHandler` to credit accounts. Arguably, this logic could have remained in `PaymentHandler`. However, `PaymentHandler` has no direct dependency or interaction with `User`, and it seemed prudent to maintain this separation and delegate the responsibility back to the `AppHandler`, which already has extensive interaction with `User`.

Example 2-1. Tweet Relevance—Product.py

```
class Product(object):

  @staticmethod
  def getProduct():

    return {'price' : 9.99, 'quantity' : 30, 'units' : 'days'}
```

Example 2-2. Tweet Relevance—User.py

```
from google.appengine.ext import db
```

```
class User(db.Model):
  twitter_username = db.StringProperty(required=True)

  # Set to "now" the first time the instance is added to the datastore
  access_start = db.DateTimeProperty(required=True, auto_now_add=True)

  # Days
  access_duration = db.IntegerProperty(required=True, default=30)
```

Example 2-3. Tweet Relevance—methods added to AppHandler.py

```
  @staticmethod
  def creditUserAccount(twitter_username, num_days):
    query = User.all().filter("twitter_username =", twitter_username)
    user = query.get()
    user.access_start = datetime.datetime.now()
    user.access_duration = num_days
    db.put(user)


  @staticmethod
  def accountIsCurrent(user):
      days_used = (datetime.datetime.now() - user.access_start).days + 1
      return days_used < user.access_duration
```

With an understanding of `Product` and `User` in place, let's now turn to the subject of integrating payment-specific logic via `PaymentHandler`. Example 2-4 illustrates an updated *main.py* file that includes a new reference to `PaymentHandler` that services APIs and encapsulates payment logic. Recall that `PaymentHandler` itself takes care of the web application logic associated with callbacks for an Express Checkout; internally, it will reference a `paypal` module that will handle the specific APIs, such as `SetExpressCheckout`, `GetExpressCheckoutDetails`, and `DoExpressCheckoutPayment`.

Example 2-4. Tweet Relevance—main.py

```
# Minimal GAE imports to run the app

from google.appengine.ext import webapp
from google.appengine.ext.webapp import util

# Logic for implementing Express Checkout

from handlers.PaymentHandler import PaymentHandler

# Logic for the app itself

from handlers.AppHandler import AppHandler

# Logic for interacting with Twitter's API and serving up data, etc.

def main():

  application = webapp.WSGIApplication([
                                        # PaymentHandler URLs

                                        ('/(set_ec)', PaymentHandler),
                                        ('/(get_ec_details)', PaymentHandler),
                                        ('/(do_ec_payment)', PaymentHandler),
                                        ('/(cancel_ec)', PaymentHandler),

                                        # AppHandler URLs

                                        ('/(app)', AppHandler),
                                        ('/(data)', AppHandler),
                                        ('/(login)', AppHandler),
                                        ('/', AppHandler)
                                       ],

                                       debug=True)
  util.run_wsgi_app(application)

if __name__ == '__main__':
  main()
```

To keep the project structure nice and tidy, the `AppHandler` and `PaymentHandler` classes are maintained in separate files (*AppHandler.py* and *PaymentHandler.py*) and reside in a directory called *handlers*. In Python parlance, we'd say that we have a `handlers` package that contains `AppHandler` and `PaymentHandler` modules. Note that there was no restriction to necessarily separate out the classes into different files, to name the files the same as the classes, or to even maintain them outside of *main.py* at all. These decisions are simply one possible way to organize and maintain the source code.

As shown in Example 2-4, the logic for encapsulating the Express Checkout-related integration points is encapsulated by the `PaymentHandler` class, which serves the following URL requests:

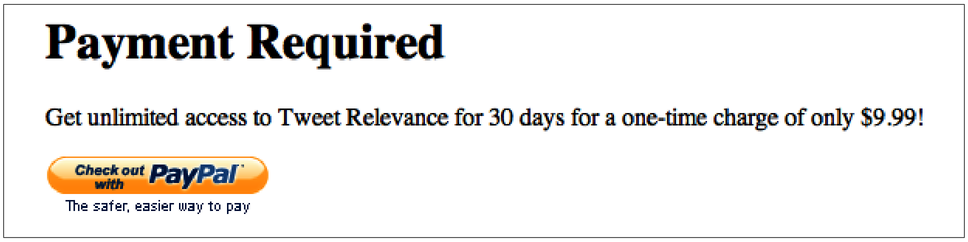
/set_ec
: A POST request to this URL sets up the transaction by calling `SetExpressCheckout` with the minimal required parameters to get a session token: the seller's 3-token credentials, the payment amount, and the return URLs that PayPal should use when redirecting users back to your website depending on whether they cancel or complete the transaction. Assuming that an OK (HTTP 200) response is returned from `SetExpressCheckout`, a session token is returned in the response, and the application immediately redirects the buyer to PayPal for completion of the purchase. The specific URL that is used for redirection within a sandbox context is https://www.sandbox.paypal.com/webscr?cmd=_express-checkout&token=xxx. If the purchase is completed successfully, PayPal redirects the buyer back to this application's `/get_ec_details` or `/cancel_ec` URLs (as defined by the `RETURNURL` and `CANCELURL` values, respectively.) Figure 2-7 displays the jumping entry point that invokes `/set_ec` and Figures 2-8 and 2-9 display the user interface provided by PayPal as part of an Express Checkout.

The minimal implementation of Express Checkout that's presented in Example 2-5 doesn't pass in additional NVP parameters such as `L_PAYMENTREQUEST_`*m*`_NAME`*n* to display details associated with an order summary and opts to use `GetExpressCheckoutDetails` to display a confirmation on your site. In Chapter 3, however, you'll implement an Express Checkout (for Digital Goods) that displays the order details on PayPal's site and bypasses the `GetExpressCheckoutDetails` on your site to accomplish a more streamlined in context payment flow.

/get_ec_details
: This URL is accessed as a GET request and provides a confirmation page summarizing the purchase and is accessed by means of PayPal redirecting the buyer back to the application once the purchase has been approved. Although not technically required, it's a best practice to call `GetExpressCheckoutDetails`, and Tweet Relevance accesses it and uses the values it returns to illustrate how to provide a confirmation page before completing the payment flow. Figure 2-10 displays a confirmation page generated from information provided by way of `/get_ec_details`.

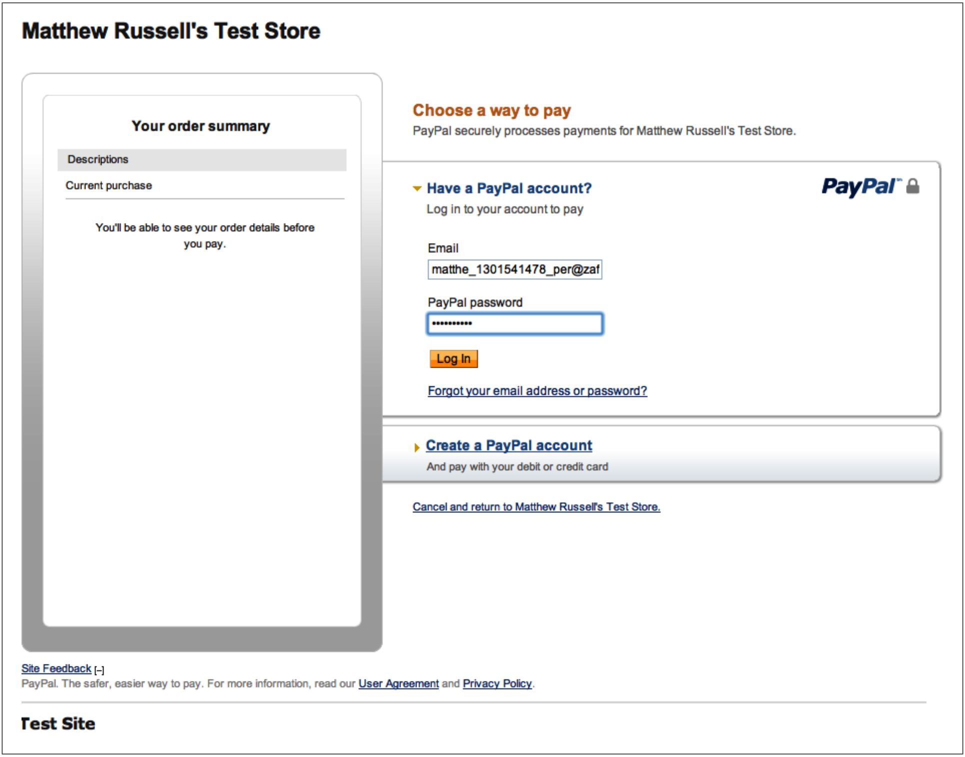
/do_ec_payment
: This URL is accessed as a GET request when the buyer approves the purchase from the confirmation page provided when `/get_ec_details` is accessed (which is after PayPal successfully redirects back to it). It finalizes the purchase by executing `DoExpressCheckoutPayment` to finalize the payment and then interfaces with `AppHandler` to credit a user's account with login tokens. Figures 2-11 displays a successful payment confirmation page generated via `/do_ec_payment`.

/cancel_ec
: If a user cancels out of the payment flow while visiting PayPal's site to approve the purchase, PayPal redirects to this URL by means of a GET request.


Payment Required

Get unlimited access to Tweet Relevance for 30 days for a one-time charge of only $9.99!

Figure 2-7. Tweet Relevance implements the Checkout Entry Point by presenting an opportunity for the user to purchase login requests. Clicking the "Checkout with PayPal" button invokes SetExpressCheckout and initiates the checkout process. (See Steps 1 and 2 of Figure 2-6.)



Matthew Russell's Test Store

Your order summary

Descriptions

Current purchase

You'll be able to see your order details before you pay.

Choose a way to pay

PayPal securely processes payments for Matthew Russell's Test Store.

Have a PayPal account?

Log in to your account to pay

Email: matthe_1301541478_per@zaf

PayPal password: ••••••••••

Log In

Forgot your email address or password?

Create a PayPal account

And pay with your debit or credit card

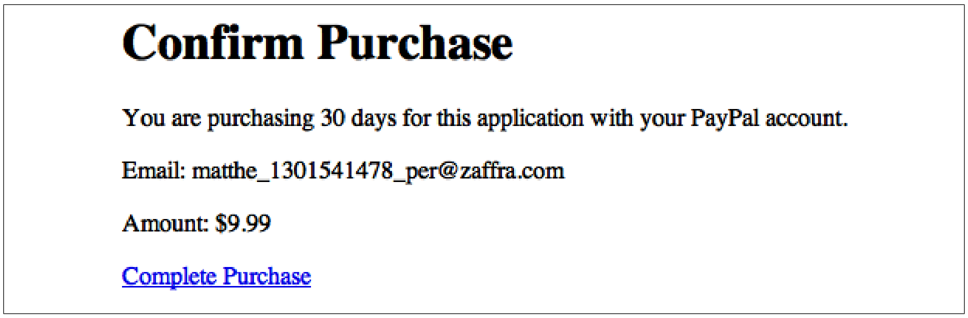
Cancel and return to Matthew Russell's Test Store.

Site Feedback [-]

PayPal. The safer, easier way to pay. For more information, read our User Agreement and Privacy Policy.

Test Site

Figure 2-8. After calling SetExpressCheckout, Tweet Relevance redirects the buyer to PayPal for payment approval. PayPal redirects buyers back to Tweet Relevance once they've authorized Tweet Relevance to charge them. (See Steps 3 and 4 of Figure 2-6.)



Confirm Purchase

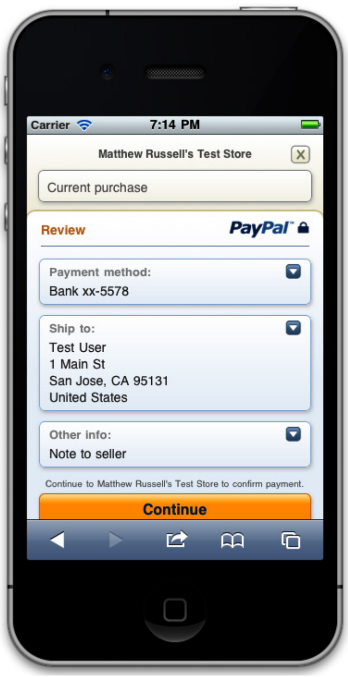
You are purchasing 30 days for this application with your PayPal account.

Email: matthe_1301541478_per@zaffra.com

Amount: $9.99

Complete Purchase

Figure 2-10. Use GetExpressDetails to present the user with a final confirmation before calling DoExpressCheckoutPayment and finalizing the transaction. (See Step 5 of Figure 2-6.)


Carrier 7:14 PM

Matthew Russell's Test Store

Current purchase

Review

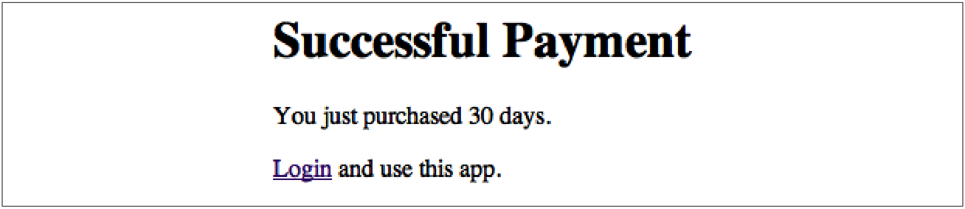
Payment method:
Bank xx-5578

Ship to:
Test User
1 Main St
San Jose, CA 95131
United States

Other info:
Note to seller

Continue to Matthew Russell's Test Store to confirm payment.

Figure 2-9. Express Checkout features a streamlined interface that's optimized for mobile devices and "just works" without any additional action required by developers.


Successful Payment

You just purchased 30 days.

Login and use this app.

Figure 2-11. Invoking DoExpressCheckoutPayment finalizes the payment and presents the user with an opportunity to log in and use the application. (See Step 6 of Figure 2-6.)

If the way that the Express Checkout was injected into the application is not fairly clear by this point, it may be helpful to reference Figure 2-6 and explicitly map the general payment flow to the APIs exposed by `PaymentHandler`. Of course, running the application's updated sample code for this chapter should be the simplest course of action to take at this point if you haven't done so already.

Finally, Examples 2-5 and 2-6 introduce the payment-related details associated with the sample code for this chapter. As is the case with the templates referenced in `AppHandler`, the templates referenced in `PaymentHandler` are nothing more than *very* minimal HTML pages that plug in named variables so that the view and controller of the application can be separated. Although there's a lot of code in these listings, it's not very complex. Example 2-5 is routing the various URL requests associated with the `PaymentHandler`'s Express Checkout operations through to PayPal, checking response values, and taking appropriate actions. Example 2-6 is a minimal class definition to create an abstraction for interacting with the Express Checkout product's API operations so that the `PaymentHandler` code is tidy and isn't littered with setting and transacting `urlfetch.fetch` operations.

Use a command-line tool such as `diff` or a text editor that's capable of producing a side-by-side diff of files to narrow in the exact changes that were made to the baseline Tweet Relevance application in order to implement Express Checkout.

Example 2-5. Tweet-Relevance/handlers/PaymentHandler.py

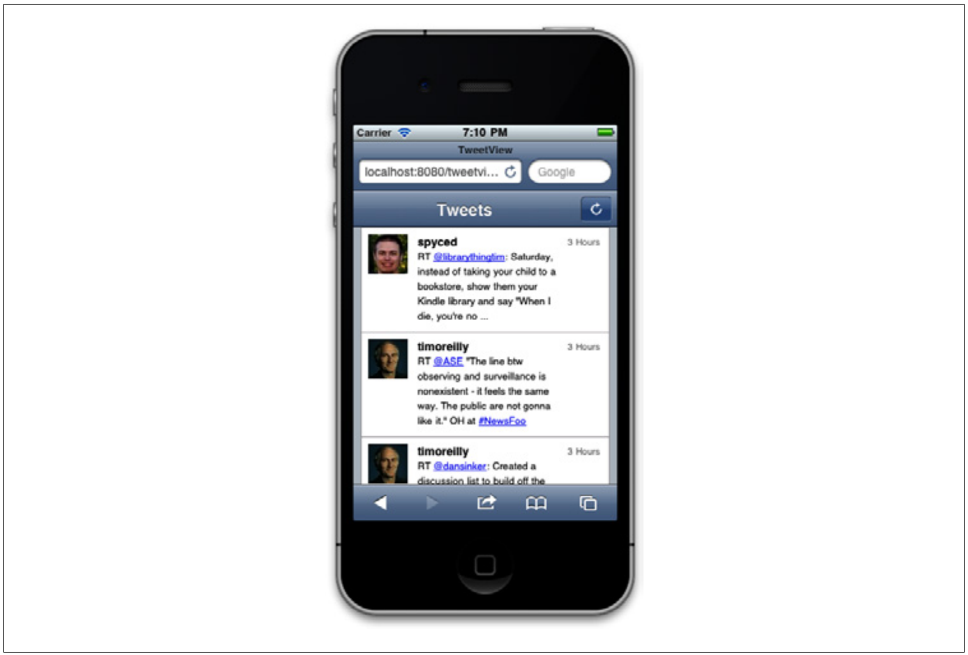
```
import os

from google.appengine.ext import webapp
from google.appengine.api import memcache
from google.appengine.ext.webapp import template
import logging
import cgi

from paypal.products import ExpressCheckout as EC
from Product import Product
from handlers.AppHandler import AppHandler

class PaymentHandler(webapp.RequestHandler):

  def post(self, mode=""):

    if mode == "set_ec":

      sid = self.request.get("sid")
      user_info = memcache.get(sid)

      product = Product.getProduct()

      nvp_params = {
              'PAYMENTREQUEST_0_AMT' : str(product['price']),
              'RETURNURL' : self.request.host_url+"/get_ec_details?sid="+sid,
              'CANCELURL': self.request.host_url+"/cancel_ec?sid="+sid
            }

      response = EC.set_express_checkout(nvp_params)

      if response.status_code != 200:
        logging.error("Failure for SetExpressCheckout")

        template_values = {
          'title' : 'Error',
          'operation' : 'SetExpressCheckout'
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
        return self.response.out.write(template.render(path, template_values))

      # Redirect to PayPal and allow user to confirm payment details.
      # Then PayPal redirects back to the /get_ec_details or /cancel_ec endpoints.
      # Assuming /get_ec_details, we complete the transaction with
PayPal.get_express_checkout_details
      # and PayPal.do_express_checkout_payment

      parsed_qs = cgi.parse_qs(response.content)

      redirect_url = EC.generate_express_checkout_redirect_url(parsed_qs['TOKEN'][0])
      return self.redirect(redirect_url)

    else:
      logging.error("Unknown mode for POST request!")

  def get(self, mode=""):

    if mode == "get_ec_details":
      response = EC.get_express_checkout_details(self.request.get("token"))

      if response.status_code != 200:
        logging.error("Failure for GetExpressCheckoutDetails")

        template_values = {
          'title' : 'Error',
          'operation' : 'GetExpressCheckoutDetails'
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
        return self.response.out.write(template.render(path, template_values))

      product = Product.getProduct()

      parsed_qs = cgi.parse_qs(response.content)

      template_values = {
        'title' : 'Confirm Purchase',
        'quantity' : product['quantity'],
        'units' : product['units'],
        'email' : parsed_qs['EMAIL'][0],
        'amount' : parsed_qs['PAYMENTREQUEST_0_AMT'][0],
        'query_string_params' : self.request.query_string
      }

      path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'confirm_purchase.html')
      self.response.out.write(template.render(path, template_values))
    elif mode == "do_ec_payment":

      if memcache.get(self.request.get("sid")) is not None: # Without an account reference,
we can't credit the purchase
        payerid = self.request.get("PayerID")

        product = Product.getProduct()

        nvp_params = {
                'PAYERID' : payerid,
                'PAYMENTREQUEST_0_AMT' : str(product['price'])
        }

        response = EC.do_express_checkout_payment(
                        self.request.get("token"),
                        nvp_params
                   )

        if response.status_code != 200:
          logging.error("Failure for DoExpressCheckoutPayment")

          template_values = {
            'title' : 'Error',
            'operation' : 'DoExpressCheckoutPayment'
          }

          path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
          return self.response.out.write(template.render(path, template_values))


        # Ensure that the payment was successful

        parsed_qs = cgi.parse_qs(response.content)

        if parsed_qs['ACK'][0] != 'Success':
          logging.error("Unsuccessful DoExpressCheckoutPayment")

          template_values = {
            'title' : 'Error',
            'details' : parsed_qs['L_LONGMESSAGE0'][0]
          }

          path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unsuccessful_payment.html')
          return self.response.out.write(template.render(path, template_values))

        if parsed_qs['PAYMENTINFO_0_PAYMENTSTATUS'][0] != 'Completed': # Probably an eCheck
          logging.error("Unsuccessful DoExpressCheckoutPayment")
          logging.error(parsed_qs)

          template_values = {
            'title' : 'Error',
            'details' : 'Sorry, eChecks are not accepted. Please send an instant payment.'
          }
          path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unsuccessful_payment.html')
          return self.response.out.write(template.render(path, template_values))


        # Credit the user's account

        user_info = memcache.get(self.request.get("sid"))
        twitter_username = user_info['username']
        product = Product.getProduct()

        AppHandler.creditUserAccount(twitter_username, product['quantity'])

        template_values = {
          'title' : 'Successful Payment',
          'quantity' : product['quantity'],
          'units' : product['units']
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'successful_payment.html')
        self.response.out.write(template.render(path, template_values))

      else:
        logging.error("Invalid/expired session in /do_ec_payment")

        template_values = {
          'title' : 'Session Expired',
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'session_expired.html')
        self.response.out.write(template.render(path, template_values))

    elif mode == "cancel_ec":
      template_values = {
        'title' : 'Cancel Purchase',
      }

      path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'cancel_purchase.html')
      self.response.out.write(template.render(path, template_values))
```

Example 2-6. Tweet-Relevance/paypal/products.py

```
from google.appengine.api import urlfetch

import urllib
import cgi

import paypal_config

def _api_call(nvp_params):

    params = nvp_params.copy()            # copy to avoid mutating nvp_params with update()
    params.update(paypal_config.nvp_params) # update with 3 token credentials and api
version

    response = urlfetch.fetch(
                paypal_config.sandbox_api_url,
                payload=urllib.urlencode(params),
                method=urlfetch.POST,
                validate_certificate=True,
                deadline=10 # seconds
               )

    if response.status_code != 200:
        decoded_url = cgi.parse_qs(result.content)

        for (k,v) in decoded_url.items():
            logging.error('%s=%s' % (k,v[0],))

        raise Exception(str(response.status_code))

    return response


class ExpressCheckout(object):

    @staticmethod
    def set_express_checkout(nvp_params):
        nvp_params.update(METHOD='SetExpressCheckout')
        return _api_call(nvp_params)

    @staticmethod
    def get_express_checkout_details(token):
        nvp_params = {'METHOD' : 'GetExpressCheckoutDetails', 'TOKEN' : token}
        return _api_call(nvp_params)

    @staticmethod
    def do_express_checkout_payment(token, nvp_params):
        nvp_params.update(METHOD='DoExpressCheckoutPayment', TOKEN=token)
        return _api_call(nvp_params)

    @staticmethod
    def generate_express_checkout_redirect_url(token):
        return "https://www.sandbox.paypal.com/webscr?cmd=_express-checkout&token=%s" %
(token,)
```

Mobile Express Checkout (MEC)

Perhaps one of the most unappreciated features of Express Checkout is that it "just works" on most mobile web browsers running a recent version of Android and iOS. When used in this context, you may hear it referred to as "Mobile Express Checkout" (MEC) or "Express Checkout for Mobile." Tweet Relevance sports a user interface that's actually optimized for mobile clients, and the decision to implement an Express Checkout turns out to look like an especially smart decision when you realize that literally no additional work is required to deliver an Express Checkout experience that's optimized for mobile devices. Figure 2-12 shows Tweet Relevance as served up on a mobile device and Figures 2-13 and 2-14 demonstrate the MEC user interface.



spyced 3 Hours
RT @librarythingtim: Saturday, instead of taking your child to a bookstore, show them your Kindle library and say "When I die, you're no ...

timoreilly 3 Hours
RT @ASE "The line btw observing and surveillance is nonexistent - it feels the same way. The public are not gonna like it." OH at #NewsFoo

timoreilly 3 Hours
RT @dansinker: Created a discussion list to build off the

Figure 2-12. Tweet Relevance features a user interface that's optimized for the mobile web

See also Appendix B, which provides a brief orientation to other options for processing mobile payments with PayPal technologies.

Recommended Exercises

While the sample code demonstrates a somewhat realistic application, it's just that: sample code. There is plenty of room for expansion and improvement. Here are a few ideas you might work though if you want to expand upon it and hone your skills:

- Use a tool such as `diff` to compare the baseline Tweet Relevance project to the modified project from this chapter. On a Linux system, for example, the following options for diff produce a convenient side-by-side display on a terminal with 237 columns when executed from the root of the source tree:

  ```
  $ diff --recursive --side-by-side --suppress-common-lines --width=237 --
  exclude=*.pyc appa ch02
  ```

  As always, read the `man` page or documentation for the utility you're using to ensure that you're taking advantage of all of the features available to you.

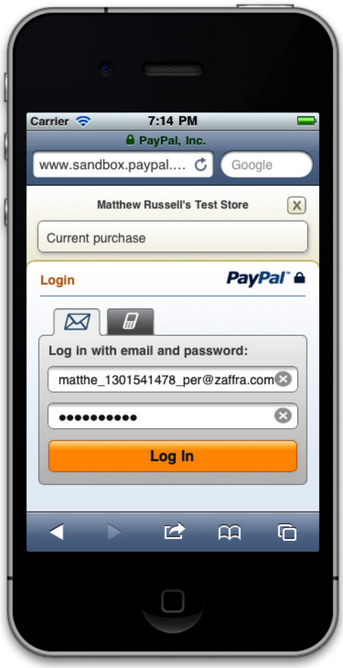
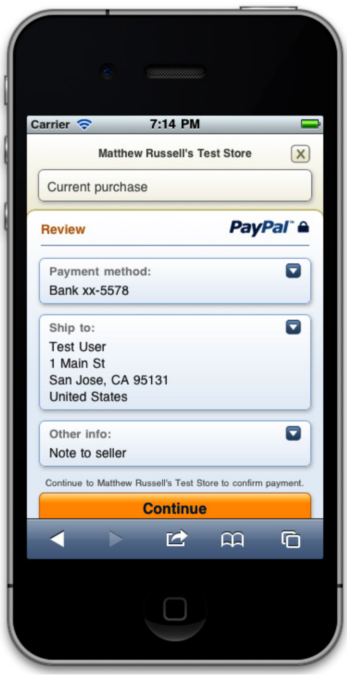


Figure 2-13. Express Checkout presents a user interface that's optimized for the mobile web when initiating a payment flow from a mobile device, making it an ideal choice for processing payments on mobile devices

- This chapter's Express Checkout flow implements only the Shopping Cart entry point. Add in the missing Payment Method.
- Modify the Express Checkout presented in this chapter so that it implements a monthly subscription model instead of a fixed-price payment model.
- Expand the available product options so that the user has the option to purchase a daily trial of the service for $0.99, a monthly subscription for $9.99, or a yearly subscription for $99.99.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!


Carrier 7:14 PM

Matthew Russell's Test Store

Current purchase

Review PayPal

Payment method:
Bank xx-5578

Ship to:
Test User
1 Main St
San Jose, CA 95131
United States

Other info:
Note to seller

Continue to Matthew Russell's Test Store to confirm payment.

Continue

Figure 2-14. Express Checkout's mobile interface makes it easy to complete the transaction

CHAPTER 3

Express Checkout for Digital Goods

This chapter is essentially a continuation of the previous one, which introduced Express Checkout. However, whereas the previous chapter established the fundamentals of Express Checkout and implemented a routine checkout for the Tweet Relevance reference application that's featured in Appendix A, this chapter works through the implementation details of a much more highly specialized checkout that can be used for situations involving "in-app" purchases of digital goods such as electronic documents, audio files, videos, or—you guessed it—items such as login tokens for Tweet Relevance using a derivative of Express Checkout called Express Checkout for Digital Goods. Of all of the PayPal products featured in this book, Express Checkout for Digital Goods is possibly the best fit for processing Tweet Relevance payments since Tweet Relevance is inherently a digital good. (Processing digital goods transactions with Adaptive Payments is probably the other best option.)

PayPal's official documentation for Express Checkout for Digital Goods is available online: Express Checkout for Digital Goods Developer Guide.

The sample code for this chapter builds upon the sample project from the previous chapter involving a traditional Express Checkout. It is highly recommended that you read that chapter first and be familiar with the implementation details in *PaymentHandler.py*.

Everyone Wins with Digital Goods Transactions

It hasn't always been the case that a Digital Goods option for Express Checkout has existed. In fact, it's a fairly new evolution of the Express Checkout product that was unveiled at PayPal's X.commerce Innovate 2010 Conference in San Francisco along with a number of other new offerings. Perhaps the most convincing reason that a specialized digital goods offering as part of the Express Checkout product makes a lot of sense is that nowadays users expect that they should be able to perform a payment without leaving their book, game, video, or application. Additionally, besides the incurred latency and inconvenience of leaving the context that can hurt conversion rates, the implementation burden that an "out and back" redirect sometimes creates for developers of context-sensitive applications such as games can be complex enough that it's nearly unbearable—at least unbearable enough that they'd rather deal with the overhead of implementing an alternative in-house payment solution than overcome the logistical barriers in interfacing with PayPal. Thus, a digital goods option serves developers and end users alike by providing the kind of in-context payment solution that users expect while minimizing the implementation burden so that developers can focus on the details of their application instead of getting derailed by the details involved in online commerce.

It's also worth calling out that digital goods are usually delivered instantly by a common form of electronic communication, and as such, there's much lower overhead incurred by suppliers, and cost savings are passed on to buyers. If you stop to think about it for a moment, you'll observe that many digital products such as books, songs, and access to "premium features" are usually priced such that they are relatively low dollar amounts. In fact, it's often the case that smart merchants will aggressively price digital goods according to common sweet spots on the spectrum of impulsive purchasing, such as $0.99, $1.99, or $2.99, in much the same way that fast-food chains price a la carte menu items at these very same price points. As such, PayPal offers a special *micropayment* transaction fee of $0.05 plus 5% for digital goods purchases up to $12.00, at which point, standard rates become more economical and apply. For example, on a $1.00 digital goods purchase, micropayment pricing incurs a fee of $0.10 as compared to the standard fee of $0.33 ($0.30 plus $2.9%), which is a savings of 70% on a $1.00 transaction! Table 3-1 shows the relative savings for micropayment rates versus standard transaction rates.

Digital goods purchases are by their very nature "instant," and as such require instant payment methods. Purchasers must use funds available from their PayPal account or use a credit card to make digital goods purchases. eChecks are not an option since they can take days to clear. See "Instant Payments versus eChecks" on page 31 for a brief overview of eChecks.

Table 3-1. Standard versus micropayment rates and associated savings as of December 2011.

Transaction Amount	Standard Rate	Micropayment Rate	Savings
$1.00	$0.33	$0.10	$0.23
$2.00	$0.36	$0.15	$0.21
$3.00	$0.39	$0.20	$0.19
$4.00	$0.42	$0.25	$0.17
$5.00	$0.45	$0.30	$0.15
$6.00	$0.47	$0.35	$0.12
$7.00	$0.50	$0.40	$0.10
$8.00	$0.53	$0.45	$0.08
$9.00	$0.56	$0.50	$0.06
$10.00	$0.59	$0.55	$0.04
$11.00	$0.62	$0.60	$0.02
$12.00	$0.65	$0.65	$0.00

To sum it all up, it would appear that everyone wins in an Express Checkout for Digital Goods: the user's experience is much improved, the seller stands to save money, and the implementation burden for the developer is significantly reduced in certain circumstances such as in-game purchases.

Implementing a Digital Goods Checkout for Tweet Relevance

This section provides a general overview of an Express Checkout for Digital Goods user experience as implemented in Tweet Relevance before transitioning into a more detailed account of the changes to the previous chapter's project code that are necessary in order to implement it.

The User Experience

If you're familiar with the implementation details for an Express Checkout, you'll find that implementing an Express Checkout for Digital Goods is almost identical. The primary differences are just that you pass in some different parameters to the core API operations (`SetExpressCheckout`, `GetExpressCheckoutDetails`, and `DoExpressCheckoutPayment`) and wire some JavaScript into your page. Aside from a marginal amount of learning curve that may be incurred by way of debugging common mistakes that can happen along the way, it's really a pretty smooth transition. Figure 3-1 illustrates the checkout flow; it's worth comparing this workflow diagram to Figure 2-6 to have a good, intuitive sense for how similar they are to one another.

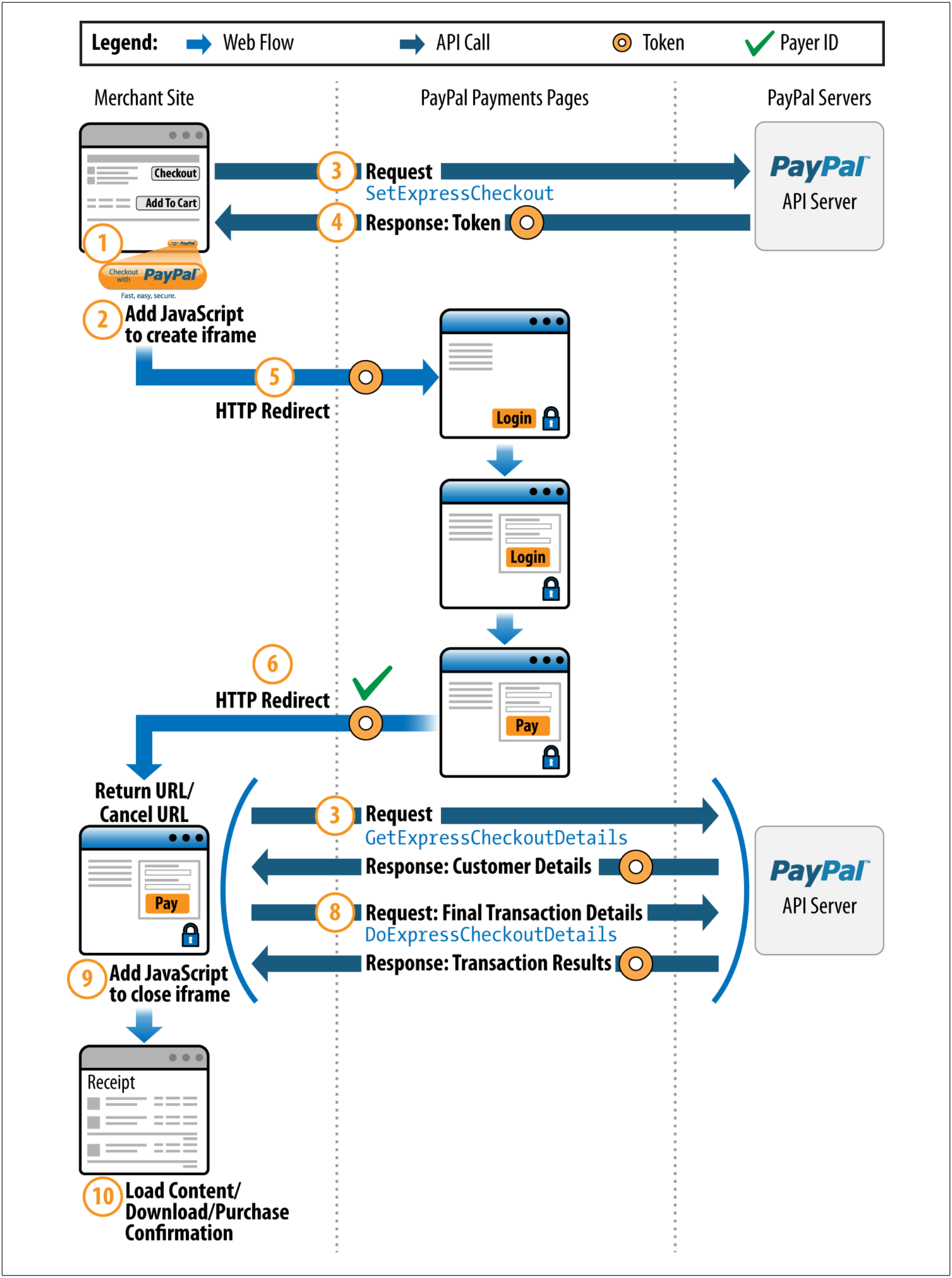

Legend:
Web Flow
API Call
Token
Payer ID
Merchant Site
PayPal Payments Pages
PayPal Servers
Checkout
Add To Cart
1
Checkout with PayPal
Fast, easy, secure.
3 Request SetExpressCheckout
4 Response: Token
PayPal API Server
2 Add JavaScript to create iframe
5 HTTP Redirect
Login
Login
6 HTTP Redirect
Pay
Return URL/ Cancel URL
Pay
3 Request GetExpressCheckoutDetails
Response: Customer Details
8 Request: Final Transaction Details DoExpressCheckoutDetails
Response: Transaction Results
PayPal API Server
9 Add JavaScript to close iframe
Receipt
10 Load Content/ Download/Purchase Confirmation

Figure 3-1. Express Checkout for Digital Goods Integration Points

The PayPal API operations that you'll use in an Express Checkout for Digital Goods are the very same as a traditional Express Checkout. A few name-value pair parameters differ in the Express Checkout API operations, and there are just a few tweaks to templates that are necessary in order to wire up a JavaScript-based, in-context user experience. Before discussing the specific changes that need to be made to the previous chapter's code, however, let's first take a closer look at the Tweet Relevance user experience as integrated with an Express Checkout for Digital Goods payment flow, as illustrated in Figures 3-2 through 3-6.

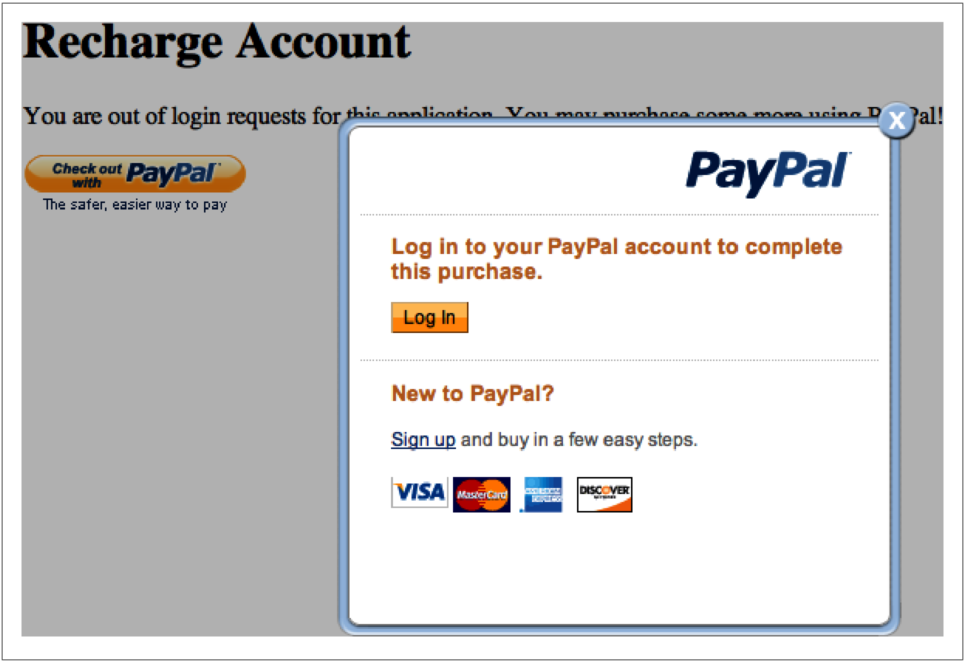
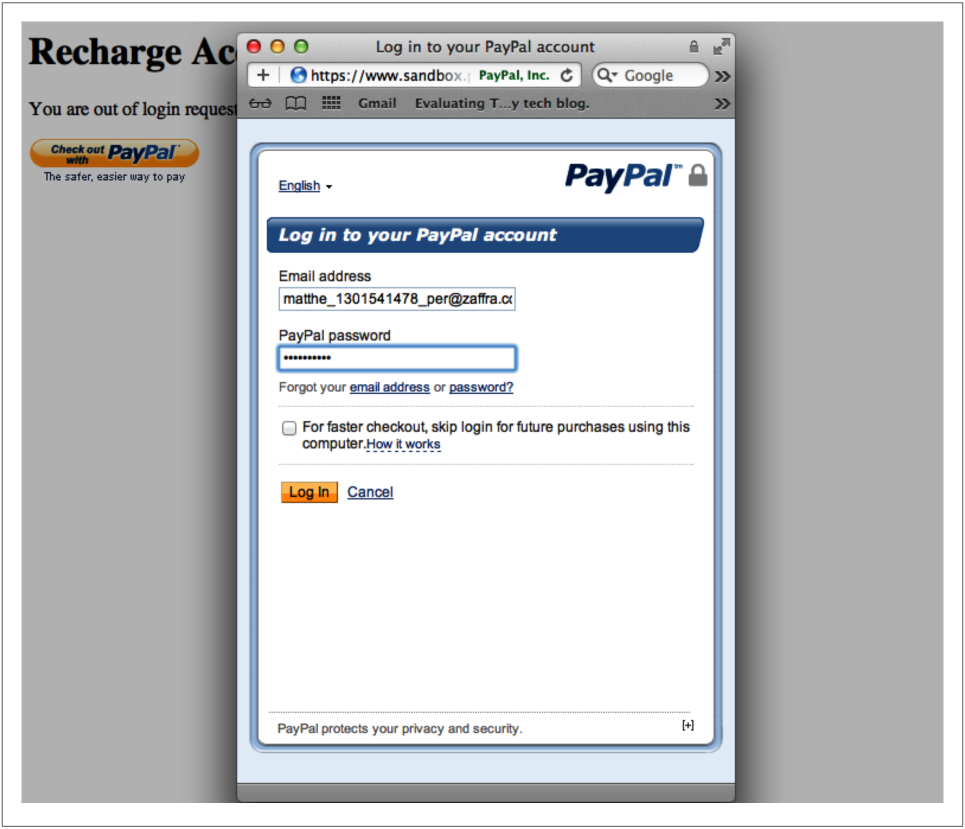

Recharge Account
You are out of login requests for this application. You may purchase some more using PayPal!
Check out with PayPal
The safer, easier way to pay
PayPal
Log in to your PayPal account to complete this purchase.
Log In
New to PayPal?
Sign up and buy in a few easy steps.

Figure 3-2. When the user clicks the PayPal button, an inline frame is spawned in the page. When the user clicks the Log In button, it spawns a new browser window that is used to initiate the login process. (See Steps 1–4 of Figure 3-1.)

Now that you have an understanding for the kind of in-context user experience that is possible with Express Checkout for Digital Goods and Tweet Relevance, let's systematically look through the changes to the project code from the previous chapter to better understand the implementation details.

Implementation Details

Perhaps the best way to navigate through the implementation details is to check out the code, explore it, and use a tool such as `diff` to identify itemized listings of the key changes, and it is highly recommended that you do so. (See "Recommended Exercises" on page 59.) A summary of these changes with relevant code snippets follows.


Recharge Ac

You are out of login request

Check out with PayPal

The safer, easier way to pay

Log in to your PayPal account

https://www.sandbox. PayPal, Inc.

Gmail Evaluating T...y tech blog.

English

PayPal

Log in to your PayPal account

Email address

matthe_1301541478_per@zaffra.cc

PayPal password

••••••••••

Forgot your email address or password?

For faster checkout, skip login for future purchases using this computer. How it works

Log In Cancel

PayPal protects your privacy and security.

Figure 3-3. Clicking the Login button from the inline frame spawns a new window so that the user can log in to PayPal and complete the purchase. At the discretion of the developer, the purchase details can be reviewed and approved at PayPal as opposed to back on your site, which is usually a more intuitive way to handle approval, given the in-context experience of the popup window. (See Steps 5-10 of Figure 3-1, noting that the Tweet Relevance flow shown here elects to take the "shortcut" approach and minimize the implementation burden.)

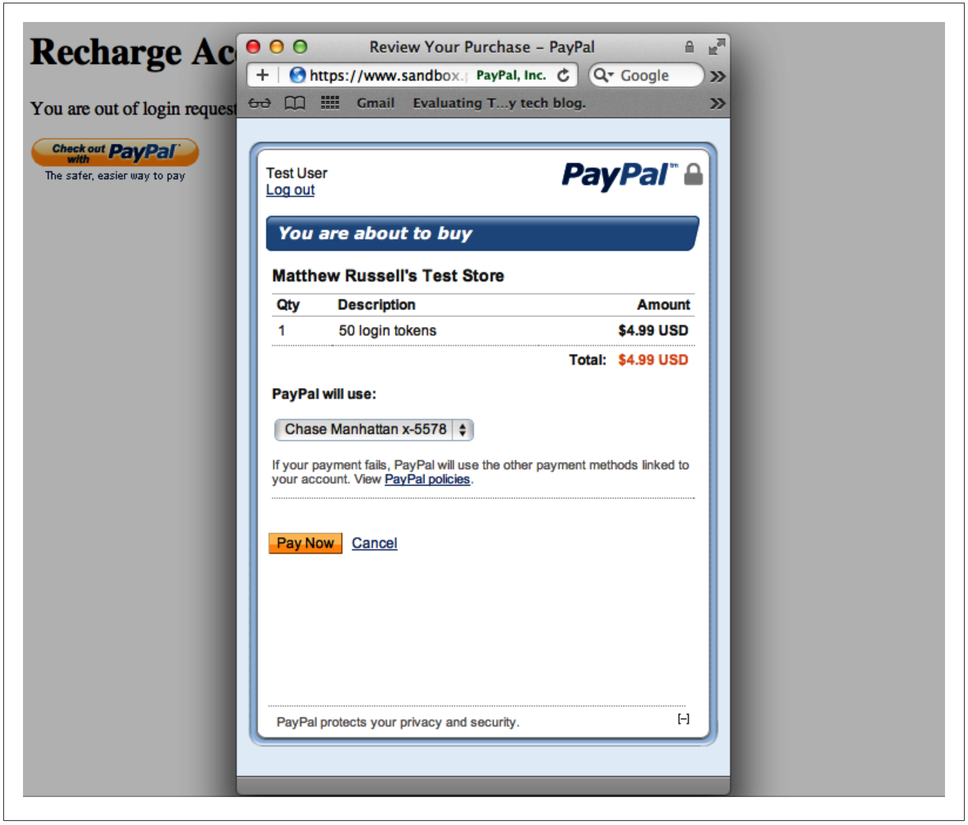
Product.py

Minimal changes are necessary to `getProduct` in order to return a simple data structure with appropriate details for a "bundle of virtual tokens" product:

```
class Product(object):

  @staticmethod
  def getProduct():
    return {'price' : 4.99, 'quantity' : 50, 'units' : 'login tokens'}
```

User.py

The `User` data model requires minimal changes so that it stores a simple counter of login requests versus an access start time and duration:


Recharge Ac
You are out of login request
Check out with PayPal
The safer, easier way to pay
Review Your Purchase – PayPal
https://www.sandbox. PayPal, Inc.
Google
Gmail
Evaluating T...y tech blog.
Test User
Log out
PayPal
You are about to buy
Pay Now
Cancel

Matthew Russell's Test Store

Qty	Description	Amount
1	50 login tokens	$4.99 USD
	Total:	$4.99 USD

PayPal will use:

Chase Manhattan x-5578

If your payment fails, PayPal will use the other payment methods linked to your account. View PayPal policies.

PayPal protects your privacy and security.

Figure 3-4. The minibrowser provides a streamlined user experience that exposes the same functionality that PayPal users have grown accustomed to.

```
class User(db.Model):
  twitter_username = db.StringProperty(required=True)
  requests_remaining = db.IntegerProperty(required=True, default=25)
```

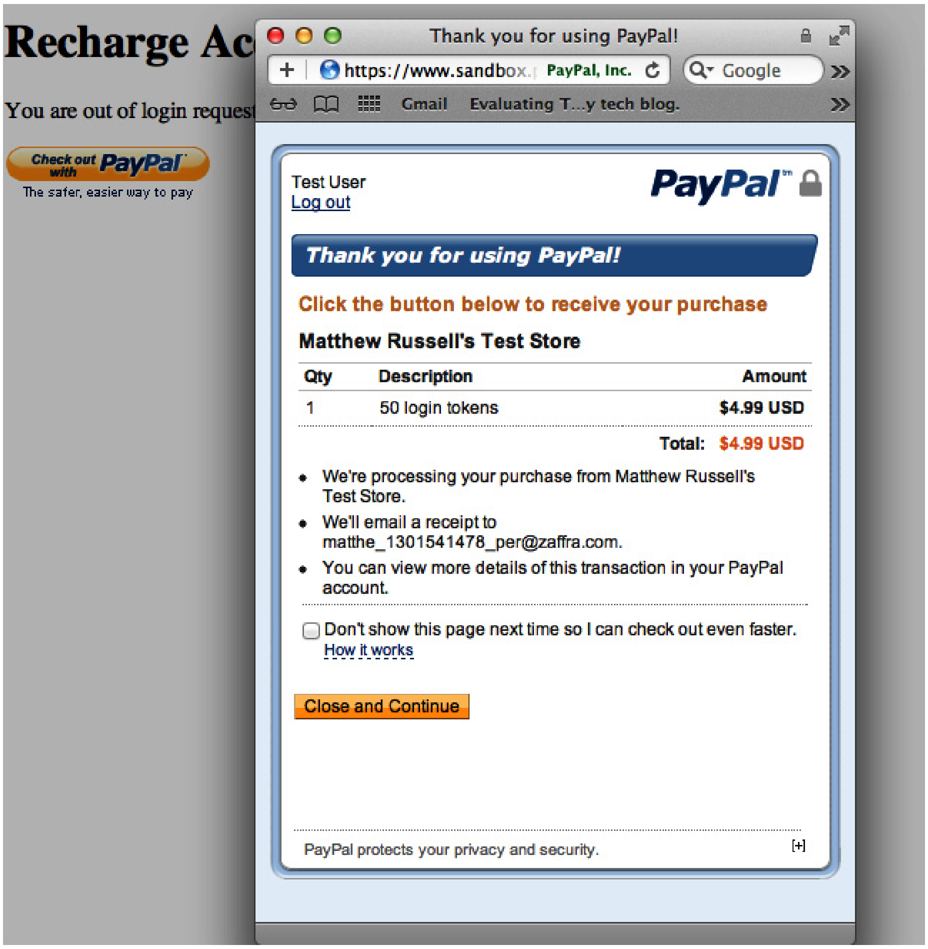
handlers/AppHandler.py

Minimal changes are necessary in order for `creditUserAccount` to inspect the `User` data model to confirm that login requests are remaining, as opposed to the calendar math that was previously involved:

```
def creditUserAccount(twitter_username, num_tokens):
  query = User.all().filter("twitter_username =", twitter_username)
  user = query.get()
  user.requests_remaining += num_tokens
  db.put(user)
```

handlers/PaymentHandler.py

Perhaps the most substantive change takes place in `PaymentHandler`. Recall that the logic for `/set_ec` sets up and initiates the checkout by executing `SetExpressCheck`


Recharge Ac

You are out of login request

Thank you for using PayPal!

https://www.sandbox. PayPal, Inc.

Test User
Log out

Thank you for using PayPal!

Click the button below to receive your purchase

Matthew Russell's Test Store

Qty	Description	Amount
1	50 login tokens	$4.99 USD
	Total:	$4.99 USD

- We're processing your purchase from Matthew Russell's Test Store.
- We'll email a receipt to matthe_1301541478_per@zaffra.com.
- You can view more details of this transaction in your PayPal account.

☐ Don't show this page next time so I can check out even faster.
How it works

Close and Continue

PayPal protects your privacy and security.

Figure 3-5. After one more click, users are back right where they left off in your application.

`out`. Here, the name-value pair parameters passed in are updated to include several important (and required) items that specify the name, amount, and quantity of the item that is presented to the buyer. Additionally, an item category is specified that indicates that the product being sold is a digital good, and the return URL is also updated to a URL that initiates the `DoExpressCheckoutPayment` API and finalizes the transaction since we are now relying on PayPal to present the item details (effectively eliminating the need for `GetExpressCheckoutDetails`). Finally, a different redirect URL is returned (more on this in a moment). The updated code for setting up the payment transaction follows with these lines emphasized:

```
def post(self, mode=""):
```

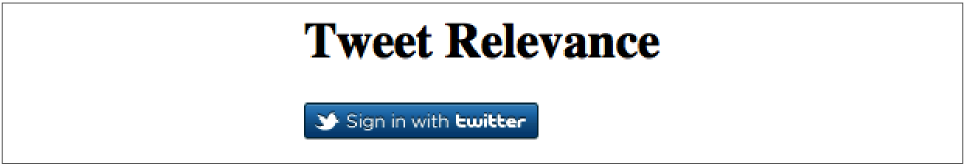

Tweet Relevance

Figure 3-6. After the user clicks the Close and Continue button on the popup window, the transaction completes, login tokens are credited to the account, and the user is returned to the Tweet Relevance login page.

```
    if mode == "set_ec":

      sid = self.request.get("sid")
      user_info = memcache.get(sid)

      product = Product.getProduct()

      nvp_params = {
              'L_PAYMENTREQUEST_0_NAME0' : str(product['quantity']) + ' ' +
product['units'],
              'L_PAYMENTREQUEST_0_AMT0' : str(product['price']),
              'L_PAYMENTREQUEST_0_QTY0' : 1,
              'L_PAYMENTREQUEST_0_ITEMCATEGORY0' : 'Digital',

              'PAYMENTREQUEST_0_AMT' : str(product['price']),
              'RETURNURL' : self.request.host_url+"/do_ec_payment?sid="+sid,
              'CANCELURL': self.request.host_url+"/cancel_ec?sid="+sid
            }

      response = EC.set_express_checkout(nvp_params)

      if response.status_code != 200:
        logging.error("Failure for SetExpressCheckout")

        template_values = {
          'title' : 'Error',
          'operation' : 'SetExpressCheckout'
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
        return self.response.out.write(template.render(path, template_values))

      # The remainder of the transaction is completed in context

      parsed_qs = cgi.parse_qs(response.content)

      redirect_url =
EC.generate_express_checkout_digital_goods_redirect_url(parsed_qs['TOKEN'][0])
      return self.redirect(redirect_url)

    else:
      logging.error("Unknown mode for POST request!")
```

The logic for `/do_ec_payment` likewise involves a minimal change so that the four `L_PAYMENTREQUEST` parameters that were mentioned can appear in the name-value pairs that are passed in with `DoExpressCheckoutPayment`. Perhaps the most important detail to call out with the addition of these parameters is that the appearance of `Digital` as the item category ensures that the improved digital goods rates in Table 3-1 apply; according to PayPal, the omission of this parameter may result in standard rates being applied.

paypal/products.py

An additional method is added to return a redirect URL specific to the in-context digital goods checkout experience. An optional query string parameter of `useraction=commit` may be added if payment details should be verified on PayPal's site (as is implemented with the Tweet Relevance sample code in this chapter) as opposed to back on your site using `GetExpressCheckoutDetails`:

```
@staticmethod
def generate_express_checkout_digital_goods_redirect_url(token, commit=True):
    if commit:
        return "https://www.sandbox.paypal.com/incontext?token=
%s&useraction=commit" % (token,)
    else:
        return "https://www.sandbox.paypal.com/incontext?token=%s" % (token,)
```

templates/checkout.html

The checkout template that features the familiar yellow "Checkout with PayPal" button requires minimal changes. An HTML ID is added to the main form's Submit button so that clicks can be intercepted by some JavaScript code that PayPal provides to trigger an in-context checkout flow. A reference to the PayPal-provided JavaScript code along with the code to wire things together is also added to the bottom of the page:

```
<html>
  <head>
      <title>{{ title }}</title>
  </head>
  <body>
      <h1>{{ title }}</h1>
      <p>{{ msg }}</p>

      <form action='/set_ec' METHOD='POST'>
      <input type='image' name='submit'
             id='submitButton'
             src='https://www.paypal.com/en_US/i/btn/btn_xpressCheckout.gif'
             border='0' align='top' alt='Check out with PayPal'/>
      <input type='hidden' name='sid' value='{{ sid }}'/>
      </form>

      <script type="text/javascript" src="https://www.paypalobjects.com/js/
external/dg.js"></script>
      <script>
          var dg = new PAYPAL.apps.DGFlow({
              trigger: "submitButton"
```

```
            });
        </script>
    </body>
</html>
```

templates/successful_payment.html

Changes to the successful payment template are pretty straightforward. After a successful payment has been completed, the updated checkout flow provides an alert to the user that the transaction is successful and redirects to the main login page:

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
        <h1>{{ title }}</h1>

        <script>
         alert('You just purchased {{ quantity }} {{ units }}. You may now login.');
          top.window.location = "/"; // Redirect to the main login page
        </script>
    </body>
</html>
```

To summarize, we've performed a very minor surgery on the sample code from the previous chapter on Express Checkout in order to implement an Express Checkout for Digital Goods payment flow. On the frontend and backend of the payment flow, a little bit of JavaScript was tactically injected to kick off and wrap up the in-context experience, and sandwiched in between were some updates to the application logic—most of which were related to changing the business model to support virtual tokens and setting up the call for `SetExpressCheckout`. In many regards, the implementation for the digital goods checkout actually seems a little bit simpler for both the developer and the buyer.

If you haven't already done so, now would be the time to pull down the sample code for this chapter and work through it in more detail. If you sell digital goods, understand the fundamentals presented in this chapter, and have some basic web development skills, you are in a great position to implement an in-context payment experience for your consumers.

Recommended Exercises

- Use a tool such as `diff` to compare the Tweet Relevance project from Chapter 2 to the modified project from this chapter. On a Linux system, for example, the following options for `diff` produce a convenient side-by-side display on a terminal with 237 columns when executed from the root of the source tree:

```
$ diff --recursive --side-by-side --suppress-common-lines --width=237 --
exclude=*.pyc ch02 ch03
```

As always, read the `man` page or documentation for the utility you're using to ensure that you're taking advantage of all of the features that you have available to you.

- Expand the product offering so that some additional bundles of virtual tokens are available at attractive prices, and implement logic so that the user is able to choose which option they'd like to purchase.
- Implement a subscription model similar to the one from the previous chapter but use a an Express Checkout for Digital Goods checkout flow instead of a traditional Express Checkout.
- Spruce up the look and feel of the application with some stylesheets and implement a more seamless login experience by "remembering the user" for a short period of time, such as 24 hours, instead of prompting for login each time that they visit the application.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!

CHAPTER 4

Adaptive Payments (Simple, Parallel, and Chained Payments)

Overview of Adaptive Payments

If you've been taking the chapter-by-chapter approach to this book, you've now learned how to implement a traditional Express Checkout payment flow as well as an Express Checkout for Digital Goods payment flow. Make no mistake that these two payment flows are great ways to implement a checkout and provide a first-class payment experience that your customers will recognize and appreciate as they make online purchases on your site. Historically, Express Checkout has been the online payment experience that most consumers have expected, and the enhancements that have enabled seamless mobile experiences and in-context payment flows for digital goods scenarios have been crucial. However, the enhancements to Express Checkout have been evolutionary steps forward and offer fairly narrow possibilities in comparison to what the Adaptive Payments product offers.

PayPal's official documentation on Adaptive Payments is available online: Adaptive Payments Developer Guide.

In comparison to Express Checkout, Website Payments Pro, and other disparate PayPal products that you may have considered as part of a checkout flow in which there is a single buyer and single seller, Adaptive Payments provides a single comprehensive framework for third parties (who are often neither the buyer nor the seller) to build payment processing systems of just about any variety you can imagine—including transactions that involve multiple recipients. For example, in addition to implementing a *simple payment* in which a single sender sends money to a single receiver, Adaptive Payments trivially allows you to use the very same API to move money from one sender to multiple receivers using what are called *chained payments* and *parallel payments*.

Handling payment preapprovals, refunds, currency conversions, and other advanced scenarios are also possible through the same flexible fabric exposed through the Adaptive Payments APIs.

Excited? The remainder of this chapter provides some of the fundamentals and then transitions into an exercise in which we'll augment Tweet Relevance to take advantage of Adaptive Payments.

Common Adaptive Payment Workflows

The adaptive Payments product allows a developer to write software that facilitates payments between a sender and one or more receivers of that payment. Unlike products such as Express Checkout where the seller is necessarily the API caller, the application (on behalf of the developer) is the caller of the Adaptive Payments API operations, so the seller and the application owner need not be the same party. The application owner must have a PayPal business-level account with the appropriate permissions levels approved by PayPal, but senders and receivers can have a PayPal account of any type or even no PayPal account since they can elect to use a "guest checkout."

Outlined in Figure 4-1, this is referred to as a simple payment, where a sender makes a payment to a single recipient. This type of payment is equivalent to what is done with Express Checkout except that the receiver of the payment is not necessarily the application developer. If you're used to the Express Checkout paradigm in which the receiver and the application owner are the same, this approach may seem a but curious. However, after some reflection, it hopefully becomes apparent that there's much to be gained by allowing a third-party developer to write applications that process payments on behalf of one or more sellers. It's not uncommon at all that a seller may want to leverage a third-party application that already exists or have a custom application built that can process payments with as little involvement in the technical development process as possible.

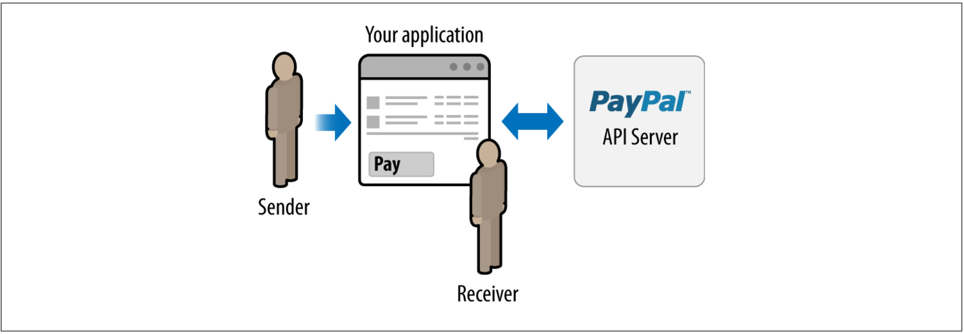


Figure 4-1. Simple Adaptive Payment

The Adaptive Payments API allows you and your application to act as an intermediary that facilitates payments for others, without you as the application developer being a recipient of the funds. As outlined in Figure 4-2, one possible scenario in which this could be the case is referred to as a *parallel payment*, in which the sender intentionally transmits a single payment to multiple recipients and has insight into how the payment is disbursed among those multiple recipients. Parallel payments are commonly used in aggregated shopping and allow a customer to order from multiple vendors with a single shopping cart.

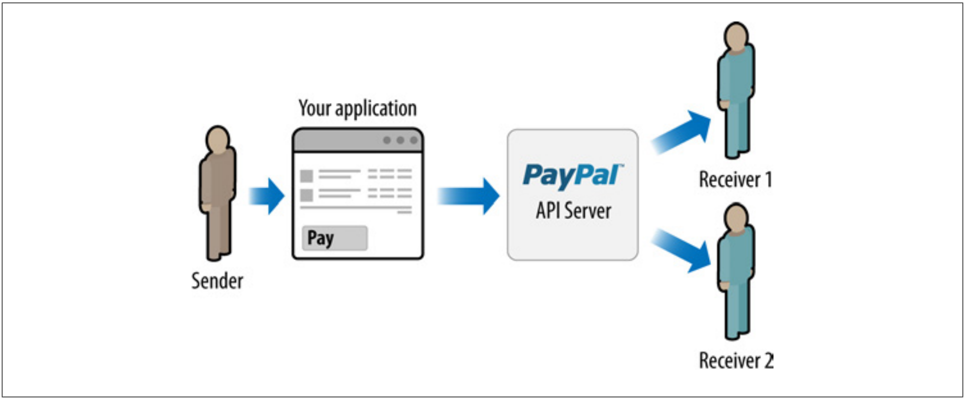


Figure 4-2. Parallel Adaptive Payment

Another way in which your application can function as an intermediary to multiple recipients with Adaptive Payments is through a *chained payment* as outlined in Figure 4-3. In a chained transaction, your application receives the payment, and the funds are then split between multiple recipients on the backend. In other words, the sender transmits funds to a single receiver and the receiver who then passes on some portion of the funds to multiple recipients. In a chained setup, your application or any other receiver could take a percentage of the payment and then disperse the remaining funds to the other recipients. For example, you might purchase an "all inclusive" vacation package from a travel site that collects a nominal fee before passing through portions of the payment to various parties involved in your vacation experience. It is even possible to set up a *delayed chained payment* that allows for a delay in the secondary receivers collecting payment. One scenario in which delayed chained payments can be handy involves secondary receivers needing to ship goods before they receive their payment for the transaction.

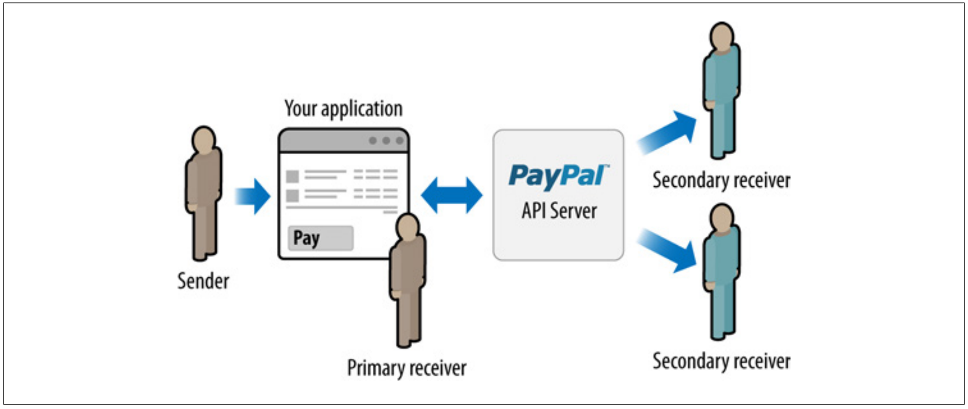


Figure 4-3. Chained Adaptive Payment

Who Pays the Fees?

Given the flexible nature of Adaptive Payments, you may very well be wondering whose responsibility it is to pay the fees for a transaction involving multiple receivers. The short answer is that there are four primary configurations that are available to distribute the fees for an Adaptive Payment transaction:

- Sender pays the fee, regardless of payment type (simple, parallel, or chained)
- All receivers pay the fee in a parallel payment
- All receivers pay the fee in a chained payment
- Only the primary receiver pays the fee in a chained payment

If you consider the various payment situations described so far in this chapter, these four possibilities really do cover the bases. In any payment scenario, it makes sense that the sender could pick up the tab for the fees on behalf of all of the receivers in the transaction just as easily as each receiver could pick up the tab for only their portion of the fees, regardless of whether the payment is simple, parallel, or chained. In any of these scenarios, there is no fundamental difference in how fees are calculated, so PayPal takes the same cut either way. However, with a chained payment, it seems reasonable that the primary receiver should be able to pick up the tab for all of the fees on behalf of the secondary receivers, so that's a possibility as well. From an implementation standpoint, the application developer is able to very easily configure who pays the fees according to a configuration parameter that's passed in when setting up the payment transaction.

Although this chapter doesn't revisit the notion of digital goods, it's also certainly possible to utilize Adaptive Payments for processing digital goods transactions, which opens up the possibility for the more cost-effective micropayments scheme for eligible transaction types.

As you are now surely realizing, the flexibility of Adaptive Payments facilitates a vast number of payment possibilities. With a high-level overview now established, let's dig a little deeper into the APIs.

Payment Approval and Payment Flows

When a payment transaction via an Adaptive Payments application has been submitted, one of four different payment approval types is involved. The remainder of this section reviews these approval types: Explicit Payments, Preapproved Payments, Implicit Payments, and Guest Payments. Although we won't use all of the approaches for payment integration with Tweet Relevance in this chapter, it's important to know that these fundamental possibilities exist.

Explicit Payments

Explicit Payments require the sender to log into PayPal.com and approve each individual payment just like in an Express Checkout payment experience. This is the traditional method for paying via PayPal and is the only option a sender has for executing a payment, unless a preapproval agreement has been established (a Preapproved Payment), or unless the sender is also the application provider (an Implicit Payment). The interaction between your application and PayPal can be controlled during the transaction process by providing URLs for redirecting the sender, depending on the situation. Figure 4-4 outlines an Explicit Payment flow, which consists of the following steps:

1. Your application sends a `Pay` request to PayPal.
2. PayPal responds with a payment key that you use to redirect the sender to PayPal.
3. You redirect the sender to PayPal.com.
4. The sender approves the transfer of the payment, and PayPal redirects the sender to a return URL.
5. PayPal sends both the receiver and the sender an email summarizing the payment that was made.

Although not explicitly listed as a step in Figure 4-4, applications should be careful to verify the status of a payment immediately after the redirect of Step 4 before taking any actions such as crediting a user account. Usually, this is accomplished by using the same key that was returned in Step 2. For example, if PayPal redirects back to http://example.com/successful_payment?user=Bob&item=123 and your application blindly ships Bob Item #123 without first confirming the payment status referenced by the Pay key from Step 2, you'd be wide open to easy hacking exploits.

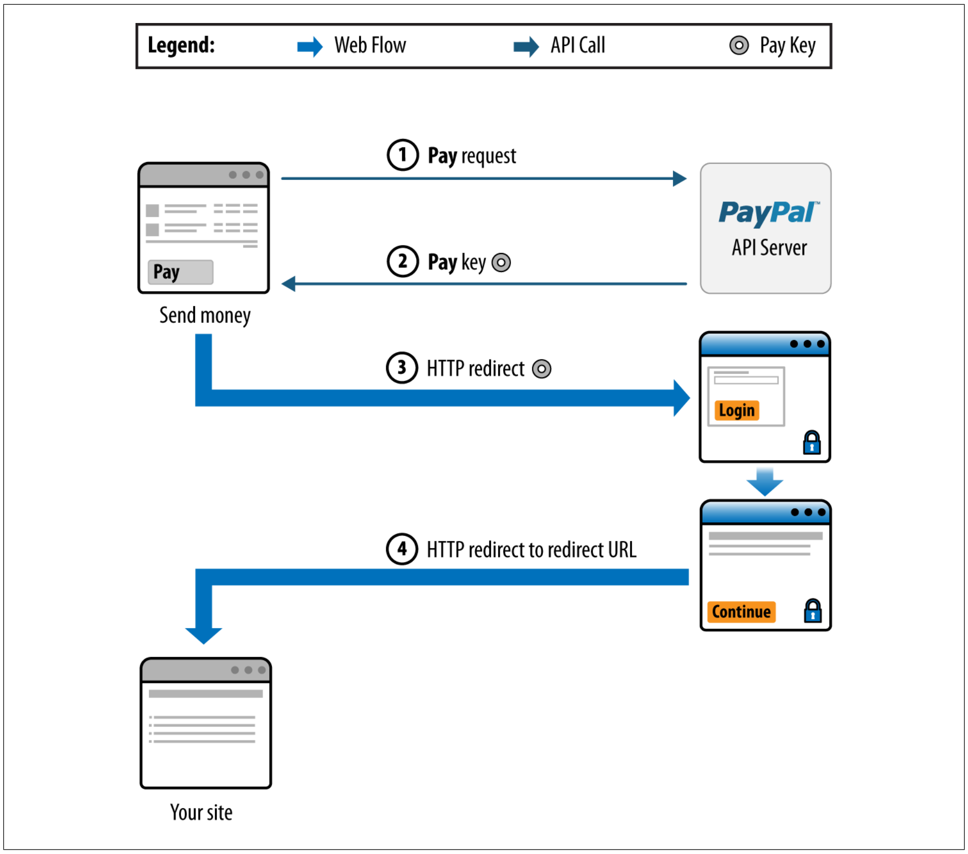


Figure 4-4. Explicit Payment flow

Preapproved Payments

Preapproved Payments allow senders to log into PayPal.com and set up preapprovals for future payments so that explicit payments are not required for every single transaction. Once the preapproval is established for certain constraints such as amount or time duration, payments are automatically considered approved, and the sender will not have to log in to approve payments to that vendor in the future until one of the preapproved conditions expires. During the preapproval setup process, the sender can specify the following constraints:

- Duration of the preapproval, including the start date and end date
- The maximum amount being approved at one time
- The maximum number of payments allowed for the vendor

Figure 4-5 outlines a Preapproved Payment flow, which consists of the following steps:

1. An application sends a preapproval request to PayPal.

2. PayPal responds with a preapproval key that you use in redirecting the sender to PayPal.
3. The application redirects the sender to PayPal.
4. After the sender approves the preapproval, PayPal redirects the sender to a return URL.
5. PayPal sends both the receiver and the sender an email summarizing the payment that was made.

Once the sender approves the preapproval setup, an application can make payments on behalf of the sender directly, as outlined in Figure 4-6.

1. Your application sends a `Pay` request to PayPal that includes a preapproval key identifying the payment agreement.
2. PayPal responds with a payment key that is used for other API functions.

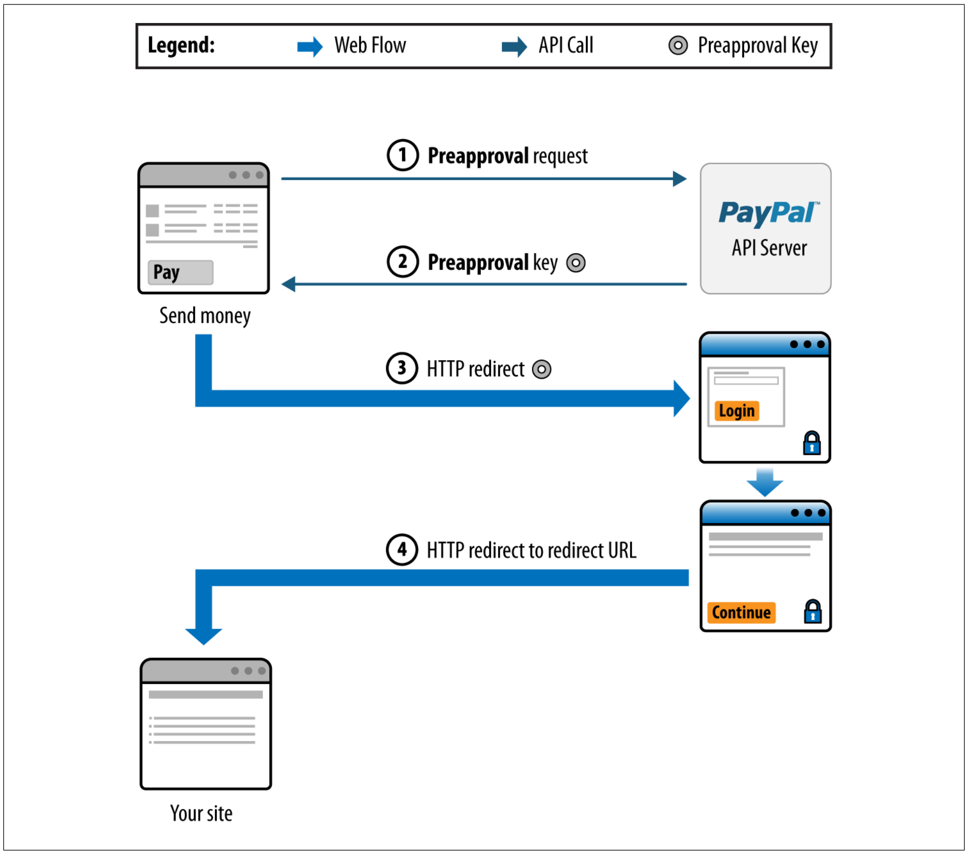


Figure 4-5. Preapproved Payment flow

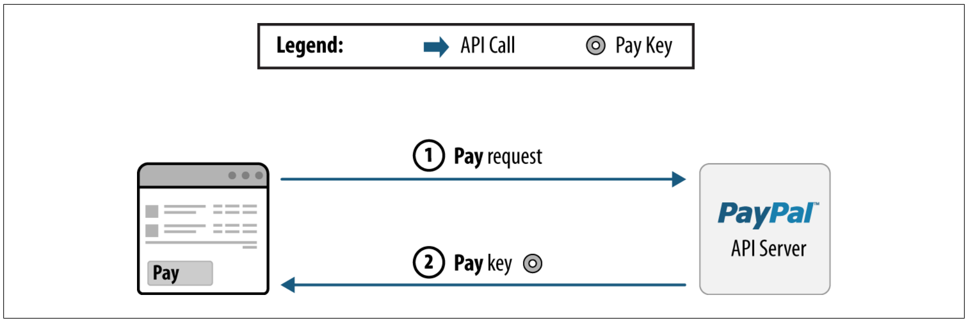


Figure 4-6. Preapproved Payment direct sending

Implicit Payments

Implicit Payments are payments sent directly by an application whose owner is also API caller. In the case of an Implicit Payment, no approval is necessary for the payment transaction since the parties are one and the same. Figure 4-7 outlines an Implicit Payment.

1. Your application sends a `Pay` request to PayPal.
2. PayPal responds with a key to use for other API operations.

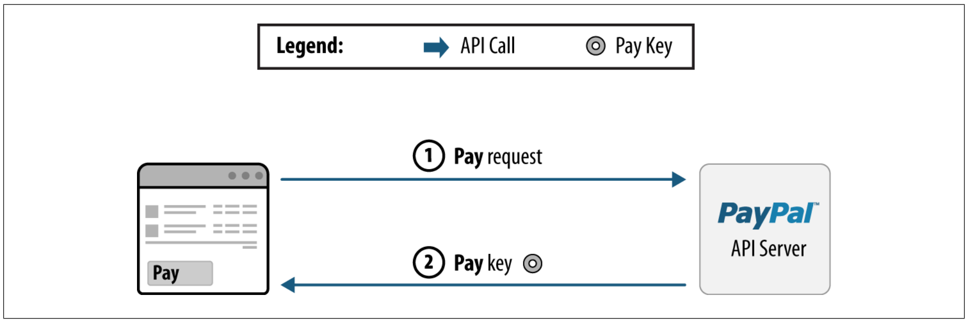


Figure 4-7. Implicit Payment flow

Guest Payments

The Adaptive Payments product also supports Guest Payments, in which the sender can pay without a PayPal account by using a credit card. Guest Payments are handled in the same manner as Explicit Payments, except that the sender provides credit card information directly on the PayPal payment screen. The recipient of a Guest Payment must have either a business- or premier-level PayPal account in order for an application to process a Guest Payment.

The Pay and PaymentDetails APIs

Before demonstrating an Adaptive Payments integration with the Tweet Relevance sample code from Appendix A, let's take a closer look at the Adaptive Payments `Pay`[1] and `PaymentDetails`[2] APIs, which are integral to our implementation details. A complete and more comprehensive list of all of the Adaptive Payments API operations can be found at Adaptive Payments API documentation on X.com. This section focuses on two of the most essential APIs and relevant options for Tweet Relevance payment integration along with some sample code to quickly get you up and running.

If you come from an Express Checkout background or have been following along in the book chapter by chapter, an important distinction to make up front between Adaptive Payments and Express Checkout is the nature of the request parameters. Whereas Express Checkout involves sending in 3-Token credentials along with Name-Value pairs through the request body, Adaptive Payments require 3-Token credentials along with a mandatory application identifier and additional configuration information to be passed in as headers; the POST request payload identifies the API operation and its parameters. Let's now turn to the Pay API and execute some sample API calls to see how it all works.

Pay API Operation

All payments made via the `Pay` API have the same essential fields and are outlined in Table 4-1. If thinking about Adaptive Payments from an Express Checkout mindset, you might consider the `Pay` API to be similar to the `SetExpressCheckout` API in that it sets up a transaction and returns a value called a "pay key" that can be used to redirect a sender to PayPal for approval.

Table 4-1. Common fields for the Pay API encoded in NVP format

Field	Description
`actionType`	Will be one of three possible values: • `PAY`: Use this value to set up a payment transaction except when using the request in combination with `ExecutePaymentRe quest`. • `CREATE`: Used to set up payment instructions with a `SetPaymen tOptions` request and then execute at a later time with an `Execu tePaymentRequest`. • `PAY_PRIMARY`: Used for chained payment situations only. This allows you to delay payments to secondary receivers at the time of the

1. *https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/e_howto_api_APPayAPI*
2. *https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/e_howto_api_APPaymentDetails*

Field	Description
	transaction and process only the primary receiver. To process the secondary payments, initiate `ExecutePaymentRequest` and pass the pay key obtained from the `PayResponse`.
`receiverList .receiver(`*n*`).email`	One or more receivers' email addresses, where n can take on values between 0 and 5. For parallel payments, up to 6 receivers may be identified, and for chained payments, 1 primary receiver and 5 secondary receivers may be identified.
`receiverList .receiver(`*n*`) .amount`	The amount to be credited to each receiver's account.
`receiverList.receiver(`*n*`).primary`	(Optional) Set this value to `true` to indicate that this is a chained payment. Only one receiver can be the primary receiver.
`currencyCode`	The code for the currency in which the payment is made. You can specify only one currency, regardless of the number of receivers. A complete list of supported currency codes is available online.
`cancelUrl`	The URL for sender redirection if the sender cancels the payment approval. This value is required, but used only for explicit payments.
`returnUrl`	The URL for sender redirection after completion of the payment. This value is required, but used only for explicit payments.
`requestEnvelope.errorLanguage`	The `requestEnvelope` is required information common to each API operation and includes members such as `errorLanguage`, the language in which error messages are displayed, and the level of detail that should be returned for error messages. At the current time, the only supported error language is US English (`en_US`).
`feesPayer`	(Optional) The payer of PayPal fees. Allowable values are: • `SENDER`: Sender pays all fees (for personal, implicit simple/parallel payments; do not use for chained or unilateral payments) • `PRIMARYRECEIVER`: Primary receiver pays all fees (chained payments only) • `EACHRECEIVER`: Each receiver pays his own fee (default, personal, and unilateral payments) • `SECONDARYONLY`: Secondary receivers pay all fees (use only for chained payments with one secondary receiver)

It's a fine detail, but do note that part of the request includes a mandatory "request envelope." It's a subtle but important point that the existence of a dot separator in field names for the Adaptive Payments APIs indicates a notion of hierarchy. For example, `requestEnvelope.errorLanguage` connotes that there's a `requestEnvelope` field with a sub-field `errorLanguage`. As you'll see later in this chapter, a JSON object expressing this same field would be `{'requestEnvelope' : {'errorLanguage' : 'en_US'}}`. Additional parameters are possible to include as part of the request envelope and are specific to particular Adaptive Payments API operations and indicated in the more comprehensive online documentation.

For readers familiar with a Linux or Unix shell, a trivial Bash script that uses the `curl` command to execute a request might look like Example 4-1. Readers unfamiliar with Bash or command-line utilities should simply focus on the structure of the `curl` command that is being executed. A brief explanation follows, and subsequent examples for this chapter are written in Python as GAE web applications, so there's no need to fret if learning Bash syntax wasn't part of your expectations for this chapter.

Example 4-1. Bash script demonstrating execution of the Pay API

```
#!/bin/bash

USERID="XXX"
PASSWORD="XXX"
SIGNATURE="XXX"

APPID="APP-80W284485P519543T"

RECEIVER="XXX"

AMOUNT="1.00"
CANCELURL="http://example.com/cancel"
RETURNURL="http://example.com/return"

RESULT=$(curl -s --insecure \
-H "X-PAYPAL-SECURITY-USERID: $USERID" \
-H "X-PAYPAL-SECURITY-PASSWORD: $PASSWORD" \
-H "X-PAYPAL-SECURITY-SIGNATURE: $SIGNATURE" \
-H "X-PAYPAL-REQUEST-DATA-FORMAT: NV" \
-H "X-PAYPAL-RESPONSE-DATA-FORMAT: JSON" \
-H "X-PAYPAL-APPLICATION-ID: $APPID" \
https://svcs.sandbox.paypal.com/AdaptivePayments/Pay -d
"requestEnvelope.errorLanguage=en_US\
&actionType=PAY\
&receiverList.receiver(0).email=$RECEIVER\
&receiverList.receiver(0).amount=$AMOUNT\
&currencyCode=USD\
&feesPayer=EACHRECEIVER\
&memo=Simple payment example.\
&cancelUrl=$CANCELURL\
&returnUrl=$RETURNURL\
;)
```

```
echo $RESULT
```

In short, the script sets up a few variables, executes a `curl` command using those variables along with some other parameters, and displays the results. Although it's just a trivial script, there's a lot that can be gleaned. The following observations may be helpful in solidifying your understanding of how an Adaptive Payments `Pay` API operation takes place:

- The `USERID`, `PASSWORD`, and `SIGNATURE` are the 3-Token credentials associated with the PayPal developer account for the application making this request.
- The `APPID` shown in the script is the global and shared application identifier for development purposes. (You'd request an application identifier for production use separately from PayPal when your application is ready to go live.)
- The remaining variables should look familiar: there's a receiver (who may or may not be the same as the application owner), a purchase amount, and URLs that PayPal uses to redirect the sender back to your site depending on whether or not the purchase was completed or cancelled. The sender is not identified in this request, but the sender's identify will become known once the sender logs into PayPal to approve the request.
- 3-Token credentials, request and response formats, and the application identifier are passed in as headers via `curl`'s `-H` option.
  - The request format is in NVP format, as indicated by `NV`
  - The response format is returned in JSON format as indicated by `JSON`
- The Pay request is routed to `https://svcs.sandbox.paypal.com/AdaptivePayments/Pay`, which is the Sandbox URL for the `Pay` operation, and the POST request payload as encoded in name-value pairs follows the `-d` option.
- There's a single recipient (as identified by the `receiverList.receiver(0)` values) that indicates that this Adaptive Payments transaction is a simple payment and this single recipient is footing the fees for the transaction as indicated by the `EACHRECEIVER` value.

A sample response from executing the previous `Pay` API operation follows:

```
{ "payKey" : "AP-54G358058T2731358",
  "paymentExecStatus" : "CREATED",
  "responseEnvelope" : { "ack" : "Success",
      "build" : "2428464",
      "correlationId" : "7ca7e3aa6a999",
      "timestamp" : "2012-01-14T15:36:31.515-08:00"
    }
}
```

In short, the response is formatted as JSON as requested by the `X-PAYPAL-RESPONSE-DATA-FORMAT` header, a response envelope returns an acknowledgment that the request is successful, and the response indicates that a payment request has been created and includes a `payKey` value that can be used to redirect a sender to PayPal for approval. To initiate the approval process for a Sandbox application, an application must redirect the sender back to *https://www.sandbox.paypal.com/cgi-bin/webscr?cmd=_ap-payment&paykey=*`value`.

Although not germane to the Tweet Relevance integration, it's worthwhile to note that if the API caller and the sender are one and the same (an Implicit Payment), a `senderEmail` field can be specified, and PayPal will implicitly approve the payment without redirecting to PayPal for explicit approval. You can also use a preapproval to execute the payment and avoid explicit approval. The required preapproval fields include a preapproval key and personal identification number (PIN).

The PaymentDetails API Operation

The `PaymentDetails` API is used to obtain information about a payment. You can identify the payment by your tracking ID, the PayPal transaction ID in an IPN message, or the pay key associated with the payment. Table 4-2 summarizes the common request parameters.

Table 4-2. Common PaymentDetails request fields

Field	Descriptions
`payKey`	This field identifies the payment for which you wish to set up payment options. This is the key that is returned in the `PayResponse` message.
`requestEnvelope.errorLanguage`	The `requestEnvelope` is required information common to each API operation and includes members such as `errorLanguage`, the language in which error messages are displayed, and the level of detail that should be returned for error messages.
`transactionId`	(Optional) The PayPal transaction ID associated with the payment. The IPN message associated with the payment contains the transaction ID.
`trackingId`	(Optional) The tracking ID that was specified for this payment in the PayRequest message. Maximum length: 127 characters.

In short, you pass in one of several possible values that identifies a payment to `PaymentDetails`, and it returns relevant status information about the payment. Example 4-2 illustrates a trivial Bash script that makes a `PaymentDetails` API request using a `payKey` value returned from Example 4-1. Example usage for the script is to simply pass in the pay key as a command-line parameter to the script.

Example 4-2. Bash script illustrating usage of the PaymentDetails API

```
#!/bin/bash

PAYKEY="${1}"

USERID="XXX"
```

```
PASSWORD="XXX"
SIGNATURE="XXX"

APPID="APP-80W284485P519543T"

RESULT=$(curl -s --insecure \
-H "X-PAYPAL-SECURITY-USERID: $USERID" \
-H "X-PAYPAL-SECURITY-PASSWORD: $PASSWORD" \
-H "X-PAYPAL-SECURITY-SIGNATURE: $SIGNATURE" \
-H "X-PAYPAL-REQUEST-DATA-FORMAT: NV" \
-H "X-PAYPAL-RESPONSE-DATA-FORMAT: JSON" \
-H "X-PAYPAL-APPLICATION-ID: $APPID" \
https://svcs.sandbox.paypal.com/AdaptivePayments/PaymentDetails -d
"requestEnvelope.errorLanguage=en_US\
&payKey=$PAYKEY"\
;)

echo $RESULT
```

Sample results from the script follow and illustrate the basic format of a `PaymentDetails` response:

```
{ "actionType" : "PAY",
  "cancelUrl" : "http://example.com/cancel",
  "currencyCode" : "USD",
  "feesPayer" : "EACHRECEIVER",
  "ipnNotificationUrl" : "http://example.com/ipn",
  "memo" : "Simple payment example.",
  "payKey" : "AP-4U527241GF1114245",
  "paymentInfoList" : { "paymentInfo" : [ { "pendingRefund" : "false",
            "receiver" : { "amount" : "1.00",
                "email" : "XXX",
                "paymentType" : "SERVICE",
                "primary" : "false"
              }
          } ] },
  "responseEnvelope" : { "ack" : "Success",
      "build" : "2428464",
      "correlationId" : "4808cadb5297e",
      "timestamp" : "2012-01-14T17:58:11.358-08:00"
    },
  "returnUrl" : "http://example.com/return",
  "reverseAllParallelPaymentsOnError" : "false",
  "sender" : { "useCredentials" : "false" },
  "status" : "CREATED"
}
```

Of particular interest in the response for `PaymentDetails` is the `status` field that indicates that the payment request has been created but not yet completed; however, should you visit *https://www.sandbox.paypal.com/cgi-bin/webscr?cmd=_ap-payment&paykey=AP-4U527241GF1114245* and successfully approve the payment, invoking `PaymentDetails` again should return a `status` of `COMPLETED`. However, a `status` of `COMPLETED` does not necessarily mean that the payment was successfully processed and that payment was rendered—it only means that the request, regardless of its ultimate outcome—was completed successfully. If the `status` were `COMPLETED`, additional information would be included regarding the specific details as they relate to the payment(s). For example, the following sample results show a `PaymentDetails` response where `status` is `COMPLETED` and the `paymentInfoList` field provides definitive information about the ultimate outcome of the payment. (In the case of an eCheck payment, the `transactionStatus` would have been `PENDING`.)

```
{
  "status": "COMPLETED",
  "responseEnvelope": {
    "ack": "Success",
    "timestamp": "2012-01-31T22:47:32.121-08:00",
    "build": "2486531",
    "correlationId": "e28c831c96f87"
  },
  "returnUrl": "http://example.com",
  "payKey": "AP-72S344750E3616459",
  "senderEmail": "XXX",
  "actionType": "PAY",
  "sender": {
    "email": "matthe_1325995267_per@zaffra.com",
    "useCredentials": "false"
  },
  "paymentInfoList": {
    "paymentInfo": [
      {
        "refundedAmount": "0.00",
        "receiver": {
          "paymentType": "SERVICE",
          "amount": "9.99",
          "email": "XXX",
          "primary": "false"
        },
        "transactionId": "2NB983427X665902U",
        "senderTransactionStatus": "COMPLETED",
        "senderTransactionId": "11411689C90721011",
        "pendingRefund": "false",
        "transactionStatus": "COMPLETED"
      }
    ]
  },
  "currencyCode": "USD",
  "cancelUrl": "http://example.com/cancel",
  "feesPayer": "EACHRECEIVER",
  "reverseAllParallelPaymentsOnError": "false"
}
```

If you're comfortable working in a Linux or Unix environment or can comfortably execute `curl` commands in a Windows environment, it's worthwhile to try manually executing these scripts to ensure that you understand the fundamentals. Regardless, in the next section, we'll implement the same logic as a GAE project.

GAE Simple Adaptive Payments Example

Let's now take the concepts concerning `Pay` and `PaymentDetails` API requests from the previous sections and consolidate them into an austere GAE application. If you've been following along closely, Example 4-3 should seem fairly straightforward. It's a web app that processes requests for two URLs: 1) a request on the root context of the application that triggers a `Pay` request and displays the response, and 2) a `/status` request that executes a `PaymentDetails` request for the original `Pay` request and displays the response. Since native Python list and dictionary objects are so close to a JSON representation, it makes sense to use JSON as both the request and response format, so you'll notice a new `import` statement that makes available a JSON module for easily converting between the JSON string representation and the native Python objects. You'll also see an `import` statement that brings in the `memcache` module that's used to minimally mimic a session implementation, which the app uses to store and later look up the `payKey` returned from the `Pay` request and pass it through to the `PaymentDetails` request that is executed when `/status` is requested. Go ahead and take a look at the code; afterward, a play-by-play synopsis is provided that breaks down the action as a series of coarsely grained steps.

Example 4-3. A sample GAE application that executes a Simple Adaptive Payment—main.py

```
#!/usr/bin/env python

"""
A minimal GAE application that makes an Adaptive API request to PayPal
and parses the result. Fill in your own 3 Token Credentials and sample
account information from your own sandbox account
"""

import random

from google.appengine.ext import webapp
from google.appengine.ext.webapp import util
from google.appengine.api import urlfetch
from google.appengine.api import memcache
from django.utils import simplejson as json


# Replace these values with your own 3-Token credentials and a sample "seller"
# who is the receiver of funds to run this sample code in the developer sandbox

user_id = "XXX"
password = "XXX"
signature = "XXX"
receiver = "XXX"

class MainHandler(webapp.RequestHandler):

    # Helper function to execute requests with appropriate headers
    def _request(self, url, params):
        # standard Adaptive Payments headers
        headers = {
            'X-PAYPAL-SECURITY-USERID' : user_id,
            'X-PAYPAL-SECURITY-PASSWORD' : password,
            'X-PAYPAL-SECURITY-SIGNATURE' : signature,
            'X-PAYPAL-REQUEST-DATA-FORMAT' : 'JSON',
            'X-PAYPAL-RESPONSE-DATA-FORMAT' : 'JSON',
            'X-PAYPAL-APPLICATION-ID' : 'APP-80W284485P519543T'
        }

        return urlfetch.fetch(
                url,
                payload = json.dumps(params),
                method=urlfetch.POST,
                validate_certificate=False,
                headers=headers
               )

    def get(self, mode=""):

        # /status - executes PaymentDetails when PayPal redirects back to this app after
payment approval

        if mode == "status":

            payKey = memcache.get(self.request.get('sid'))

            params = {
              'requestEnvelope' : {'errorLanguage' : 'en_US', 'detailLevel' : 'ReturnAll'},
                'payKey' : payKey
            }

            result = self._request('https://svcs.sandbox.paypal.com/AdaptivePayments/
PaymentDetails', params)

            response = json.loads(result.content)

            if result.status_code == 200: # OK

                # Convert back to indented JSON and display it

                pretty_json = json.dumps(response,indent=2)
                self.response.out.write('<pre>%s</pre>' % (pretty_json,))
            else:
               self.response.out.write('<pre>%s</pre>' % (json.dumps(response,indent=2),))

        else: # / (application root) - executed when app loads and initiates a Pay request

            amount = 10.00

           # A cheap session implementation that's leveraged in order to lookup the payKey
         # from the Pay API and execute PaymentDetails when PayPal redirects back to /status

          sid = str(random.random())[5:] + str(random.random())[5:] + str(random.random())
[5:]
```

```
            return_url = self.request.host_url + "/status" + "?sid=" + sid
            cancel_url = return_url

            redirect_url = "https://www.sandbox.paypal.com/cgi-bin/webscr?cmd=_ap-
payment&paykey="

            params = {
                    'requestEnvelope' : {'errorLanguage' : 'en_US', 'detailLevel' :
'ReturnAll'},
                    'actionType' : 'PAY',
                    'receiverList' : {
                            'receiver' : [
                                {'email' : receiver, 'amount' : amount}
                            ],
                    },
                'currencyCode' : 'USD',
                'memo' : 'Simple payment example.',
                'cancelUrl' : cancel_url,
                'returnUrl' : return_url,
            }

           result = self._request('https://svcs.sandbox.paypal.com/AdaptivePayments/Pay',
params)

            response = json.loads(result.content)

            if result.status_code == 200: # OK

               # Convert back to indented JSON and inject a hyperlink to kick off payment
approval

                pretty_json = json.dumps(response,indent=2)
                pretty_json = pretty_json.replace(response['payKey'], '<a href="%s%s"
target="_blank">%s</a>' % (redirect_url, response['payKey'], response['payKey'],))
                memcache.set(sid, response['payKey'], time=60*10) # seconds

                self.response.out.write('<pre>%s</pre>' % (pretty_json,))
            else:
               self.response.out.write('<pre>%s</pre>' % (json.dumps(response,indent=2),))

def main():
    application = webapp.WSGIApplication([('/', MainHandler),
                                          ('/(status)', MainHandler)],
                                         debug=True)
    util.run_wsgi_app(application)

if __name__ == '__main__':
    main()
```

In terms of the overall application flow, here's how it all breaks down:

- The user requests the root context of the application.
- A session identifier is created by concatenating some random numbers together.

- A `Pay` request is executed that requires return and cancel URLs to be provided so that PayPal knows where to redirect the user after payment approval.
    - Details: We'd like for the return URL passed in with the `Pay` request to check the status of the payment associated with the `Pay` request through a subsequent `PaymentDetails` request after the user has had an opportunity to approve the payment; however, we won't have the `payKey` value that's needed for `PaymentDetails` until the `Pay` request completes, and it hasn't even been executed yet! Thus, we'll use the session identifier and specify a return URL of the form `/status?sid=123` on the `Pay` request and use `memcache` to associate the `sid` value with the `payKey` value that's returned from the `Pay` request after the `Pay` request completes.
- Results for the `Pay` request are displayed as JSON with the `payKey` hyperlinked such that the user can click on it and approve the payment, ultimately changing its status from `CREATED` to `COMPLETED`.
    - Details: The hyperlink simply involves passing in the `payKey` as a query string parameter to a standard URL of the form `https://www.sandbox.paypal.com/cgi-bin/webscr?cmd=_ap-payment&paykey=AP-808742956V333525E`.
- After the user approves payment, PayPal redirects back to the application at its return URL, which for this sample application is of the form `/status?sid=123`.
- In `/status`, the application uses the session identifier included in the URL to look up the `payKey` associated with the transaction and uses it to execute a `PaymentDetails` request. The response, whose `status` field should now be `COMPLETED`, is displayed. Recall that additional details in the response object provide definitive information regarding the ultimate outcome of the payment itself.
- Because no value for `feesPayer` is specified to override the default value of `EACHRECEIVER`, the receiver pays the fees for this transaction.

Another key point to take away from the application is that the owner of the application need not *necessarily* be the receiver of the payment. It could certainly be the case that the same owner of the application whose 3-Token credentials are supplied to run the application could also be on the receiving end, but it could just as easily be the case that the owner of the application is a third party who built the application as a fixed-price contract job and maintains the application on behalf of the receiver as part of a business arrangement. However, it's just as easily the case that perhaps the third party developer could have developed the application for free or at a deep discount in exchange for a cut of the payment. The next section illustrates how a chained payment could be used to accommodate exactly this kind of situation.

GAE Chained Adaptive Payments Example

If you understand the flow of Example 4-3, the good news is that executing a chained (or parallel payment) literally just requires a couple of additional lines of code. Recall that in a chained payment scenario, the sender perceives that a payment is being sent to a primary receiver; however, the primary receiver essentially acts as a "middle man" who takes a cut and passes on potentially variable portions of the payment to up to five additional receivers. A realistic scenario involving a chained payment could be that the developer of an application takes a cut of a payment and passes on the remaining portion to additional parties such as investors who may be stakeholders in the business venture. While a parallel payment could conceivably be used to get the money into the very same hands, a chained payment allows the additional receivers to remain anonymous so far as the sender is concerned. From the sender's point of view, there is only a single receiver.

As just mentioned, the changes to Example 4-3 that result in a chained payment are absolutely minimal. Instead of a single receiver and amount being specified and passed into the `Pay` request, multiple receivers can be passed. For example, consider the following receiver configuration:

```
params = {
        'requestEnvelope' : {'errorLanguage' : 'en_US', 'detailLevel' : 'ReturnAll'},
        'actionType' : 'PAY',
        'receiverList' : {
                'receiver' : [
                    {'email' : receiver1, 'amount' : amount1, 'primary' : True },
                    {'email' : receiver2, 'amount' : amount2, 'primary' : False},
                    {'email' : receiver3, 'amount' : amount2, 'primary' : False}
                ],
        },
    'currencyCode' : 'USD',
    'memo' : 'Chained payment example.',
    'cancelUrl' : cancel_url,
    'returnUrl' : return_url,
}
```

This configuration specifies that there is one primary receiver and two secondary receivers. If `amount1` were $10.00, `amount2` were $5.00, and `amount3` were $2.00, the primary receiver would be accepting a $10.00 payment but passing on $7.00 of it to secondary receivers—effectively taking a $3.00 cut. An important detail to also note is that because no value for `feesPayer` is specified to override the default value of `EACH RECEIVER`, all receivers, including the primary receiver, pay the fees for this transaction.

GAE Parallel Payments Example

Modifications to Example 4-3 that result in parallel payment are quite similar to those for a chained payment except that there is no designated primary receiver and the party who was the primary receiver takes an explicit cut of the payment in the parameters. Using the same configuration parameters as with the chained payment, the only code change required is that `receiver1` no longer be designated as the primary receiver. However, in order for the same payment amounts to go to the receivers in the same manner as the chained payment scenario, `amount1` would be an explicit $3.00 instead of $10.00. From the sender's point of view, there are three receivers involved with the parallel payment, and the sender has visibility into how much of the payment is given to each of the receivers.

```
params = {
        'requestEnvelope' : {'errorLanguage' : 'en_US', 'detailLevel' : 'ReturnAll'},
        'actionType' : 'PAY',
        'receiverList' : {
                'receiver' : [
                    {'email' : receiver1, 'amount' : amount1, 'primary' : False},
                    {'email' : receiver2, 'amount' : amount2, 'primary' : False},
                    {'email' : receiver3, 'amount' : amount2, 'primary' : False}
                ],
        },
    'currencyCode' : 'USD',
    'memo' : 'Parallel payment example.',
    'cancelUrl' : cancel_url,
    'returnUrl' : return_url,
}
```

As with the prior examples, all receivers pay their own portion of the fees since no value for `feesPayer` has been provided to override the default value of `EACHRECEIVER`.

Integrating a "Simple" Adaptive Payment into Tweet Relevance

If you've followed along thus far, integrating Adaptive Payments into Tweet Relevance should seem like a fairly melodramatic exercise. The goal of the integration is the same as that of previous chapters: to implement a payment mechanism so that users of the service can be charged for using it. In the interest of getting up and running, let's integrate a Simple Adaptive Payment in order to implement a basic subscription model in which a customer purchases 30 days of access for a nominal fee. The previous GAE examples have worked through most of the nuts and bolts as related to the Adaptive Payments portion of the exercise, so there's actually just a very little bit of software engineering involved to perform the integration and smooth out a few rough edges. The remainder of this section assumes that you have familiarity with the baseline Tweet Relevance project code from Appendix A and an appreciation for some of the payment models as described in "Implementing a Checkout Experience for Tweet Relevance" on page 32. Changes to the baseline project structure in order to implement a subscription payment model are addressed on a file-by-file basis.

It may be helpful to review "Implementing a Checkout Experience for Tweet Relevance" on page 32 and refresh your memory on the various payment mechanisms that could be viable for a service like Tweet Relevance. The remainder of this chapter assumes familiarity with the options as presented in that section and implements the "subscription model."

main.py

The overall architecture for the finished web application involving Adaptive Payments mimics the same operations for ExpressCheckout, but we'll name them a little differently so as not to confuse the two products. Thus, the `PaymentHander` exposes `/pay`, `/completed_payment`, and `/cancelled_payment` operations that will be mapped by the main application. Thus, `main()` looks like this:

```
def main():

  application = webapp.WSGIApplication([

                                       # PaymentHandler URLs

                                       ('/(pay)', PaymentHandler),
                                    ('/(completed_payment)', PaymentHandler),
                                    ('/(cancelled_payment)', PaymentHandler),

                                       # AppHandler URLs

                                       ('/(app)', AppHandler),
                                       ('/(data)', AppHandler),
                                       ('/(login)', AppHandler),
                                       ('/', AppHandler)
                                      ],

                                      debug=True)
  util.run_wsgi_app(application)
```

handlers/PaymentHandler.py

Most of the action for the integration happens in `PaymentHandler`, which interacts with PayPal and interfaces with the `AppHandler` to update to credit the account with 30 days of access after a successful payment. The `PaymentHandler` class in Example 4-5 illustrates how to make it happen. The `import` and reference for the trivial `Product` class is shown in Example 4-4. The basic control flow is essentially the same as that involving an Express Checkout: a transaction is set up with `/pay`, Tweet Relevance redirects to PayPal to approve the transaction, and PayPal redirects back to `/completed_payment` once the user has approved the payment. The application then confirms with PayPal that the transaction has indeed been completed before handing back control to the `ApplicationHandler` so as to avoid a fundamental security flaw in which a malicious attacker may be able to gain account credits without actually approving a payment. The same `memcache` mechanism for associating a `payKey` value with a user's session identifier after PayPal redirects back to the application, as described in Example 4-3, is also employed in `PaymentHandler`.

Finally, although it's not displayed as example code below, note that the `/pay` URL is triggered by *templates/checkout.html* in the project code—the same kind of page shown in Figure 2-7 that displays a yellow "Checkout with PayPal" button.

Example 4-4. Product.py

```
# The Product class provides product details.
# A more flexible product line could be managed in a database

class Product(object):

  @staticmethod
  def getProduct():

    return {'price' : 9.99, 'quantity' : 30, 'units' : 'days'}
```

Example 4-5. handlers/PaymentHandler.py

```
import os

from google.appengine.ext import webapp
from google.appengine.api import memcache
from google.appengine.ext.webapp import template
from django.utils import simplejson as json
import logging

from paypal.products import AdaptivePayment as AP
from paypal.paypal_config import seller_email as SELLER_EMAIL
from Product import Product
from handlers.AppHandler import AppHandler

class PaymentHandler(webapp.RequestHandler):

  def post(self, mode=""):

    if mode == "pay":

      sid = self.request.get("sid")

      returnUrl = self.request.host_url+"/completed_payment?sid="+sid,
      cancelUrl = self.request.host_url+"/cancelled_payment?sid="+sid

      product = Product.getProduct()

      seller = {'email' : SELLER_EMAIL, 'amount' : product['price']}

      response = AP.pay(receiver=[seller], cancelUrl=cancelUrl, returnUrl=returnUrl)
      result = json.loads(response.content)
      logging.info(result)

      if result['responseEnvelope']['ack'] == 'Failure':
        logging.error("Failure for Pay")

        template_values = {
          'title' : 'Error',
          'operation' : 'Pay'
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
```

```
          return self.response.out.write(template.render(path, template_values))

        # Stash away the payKey for later use

        user_info = memcache.get(sid)
        user_info['payKey'] = result['payKey']
        memcache.set(sid, user_info, time=60*10) # seconds

        # Redirect to PayPal and allow user to confirm payment details.

        redirect_url = AP.generate_adaptive_payment_redirect_url(result['payKey'])
        return self.redirect(redirect_url)

      else:
        logging.error("Unknown mode for POST request!")

    def get(self, mode=""):

      if mode == "completed_payment":

        if memcache.get(self.request.get("sid")) is not None: # Without an account
reference, we can't credit the purchase
          user_info = memcache.get(self.request.get("sid"))

          payKey = user_info["payKey"]

          response = AP.get_payment_details(payKey)
          result = json.loads(response.content)
          logging.info(result)

          if result['responseEnvelope']['ack'] == 'Failure' or \
             result['status'] != 'COMPLETED': # Something went wrong!

            logging.error("Failure for PaymentDetails")

            template_values = {
              'title' : 'Error',
              'operation' : 'ExecutePayment'
            }

            path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
            return self.response.out.write(template.render(path, template_values))


          if result['paymentInfoList']['paymentInfo'][0]['transactionStatus'] !=
'COMPLETED': # An eCheck?

            logging.error("Payment transaction status is not complete!")

            template_values = {
              'title' : 'Error',
            'details' : 'Sorry, eChecks are not accepted. Please send an instant payment.'
            }
            path = os.path.join(os.path.dirname(__file__), '..', 'templates', 
'unsuccessful_payment.html')
            return self.response.out.write(template.render(path, template_values))


          # Credit the user's account

          twitter_username = user_info['username']
          product = Product.getProduct()

          AppHandler.creditUserAccount(twitter_username, product['quantity'])

          template_values = {
            'title' : 'Successful Payment',
            'quantity' : product['quantity'],
            'units' : product['units']
          }

          path = os.path.join(os.path.dirname(__file__), '..', 'templates', 
'successful_payment.html')
          self.response.out.write(template.render(path, template_values))

        else:
          logging.error("Invalid/expired session in /completed_payment")

          template_values = {
            'title' : 'Session Expired',
          }

          path = os.path.join(os.path.dirname(__file__), '..', 'templates', 
'session_expired.html')
          self.response.out.write(template.render(path, template_values))

      elif mode == "cancelled_payment":
        template_values = {
          'title' : 'Cancel Purchase',
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates', 
'cancel_purchase.html')
        self.response.out.write(template.render(path, template_values))
```

paypal/products.py

The addition of an `AdaptivePayment` class (imported as `AP` to save some typing in `PaymentHandler`) to the `paypal.products` module, along with a few minor additions to the `paypal.paypal_config` to encapsulate configuration information such as the required Adaptive Payments headers and 3-Token credentials, are about all that it takes to round out the remainder of the substantive changes to Tweet Relevance. The `AdaptivePayment` class follows and is little more than a wrapper around a `Pay` and `PaymentDetails` request:

```
from google.appengine.api import urlfetch
from django.utils import simplejson as json
```

```
import urllib
import cgi

import paypal_config

class AdaptivePayment(object):

    @staticmethod
    def _api_call(url, params):

        response = urlfetch.fetch(
                    url,
                    payload=json.dumps(params),
                    method=urlfetch.POST,
                    validate_certificate=True,
                    deadline=10, # seconds
                    headers=paypal_config.adaptive_headers
                   )

        if response.status_code != 200:
            result = json.loads(response.content)
            logging.error(json.dumps(response.content, indent=2))

            raise Exception(str(response.status_code))

        return response


    # Lists out some of the most common parameters as keyword args. Other keyword
args can be added through kw as needed
    # Template for an item in the receiver list: {'email' : me@example.com,
'amount' : 1.00, 'primary' : False}

    @staticmethod
    def pay(sender=None, receiver=[], feesPayer='EACHRECEIVER', memo='',
cancelUrl='', returnUrl='', **kw ):

        params = {
            'requestEnvelope' : {'errorLanguage' : 'en_US', 'detailLevel' :
'ReturnAll'},
            'actionType' : 'PAY',
            'currencyCode' : 'USD',
            'senderEmail' : sender,
            'receiverList' : {
                    'receiver' : receiver
            },
            'feesPayer' : feesPayer,
            'memo' : memo,
            'cancelUrl' : cancelUrl,
            'returnUrl' : returnUrl
        }

        if sender is None: params.pop('senderEmail')

        if memo == "": params.pop('memo')
        params.update(kw)

        return
AdaptivePayment._api_call(paypal_config.adaptive_sandbox_api_pay_url, params)

    @staticmethod
    def get_payment_details(payKey):

        params = {
            'requestEnvelope' : {'errorLanguage' : 'en_US', 'detailLevel' :
'ReturnAll'},
            'payKey' : payKey
        }

        return
AdaptivePayment._api_call(paypal_config.adaptive_sandbox_api_payment_details_url,
 params)

    @staticmethod
    def generate_adaptive_payment_redirect_url(payKey, embedded=False):
        if embedded:
            return "https://www.sandbox.paypal.com/webapps/adaptivepayment/flow/
pay?payKey=%s" % (payKey,)
        else:
            return "https://www.sandbox.paypal.com/cgi-bin/webscr?cmd=_ap-
payment&paykey=%s" % (payKey,)
```

If the Adaptive Payments integration details into Tweet Relevance really do seem melodramatic, it's an indicator that your learning is well on track and that you should have little trouble using Adaptive Payments for your own application. If you haven't already, however, please take a moment to peruse the final project code as a final exercise.

There's a Lot More

Like every other chapter in this book, this chapter was designed to get you up and running—not to provide you with a comprehensive overview of Adaptive Payments. Frankly, Adaptive Payments is a such a broad, comprehensive, and exciting product that covering it in its entirety would take several hundred pages of dedicated coverage and entail writing a "definitive guide" to cover the possibilities. Using Adaptive Payments, you can quite literally handle just about any reasonable payment flow that you can imagine. At the moment, the more definitive coverage available is PayPal's Adaptive Payments Developer Guide that you can freely access online. It includes some of the same fundamentals that were introduced in this chapter but also contains examples on using embedded payment flows, preapprovals, currency conversion, issuing refunds, and more. Definitely take a little bit of time to at least peruse its table of contents.

Although we didn't cover it in this chapter, be advised that the same kinds of sophisticated payment mechanisms involved with digital goods purchases using Express Checkout are also available with Adaptive Payments. A recommended exercise for this chapter is to modify the same code for this chapter to implement an embedded payment flow.

Recommended Exercises

Some recommended exercises for furthering your knowledge of Adaptive Payments include:

- Use a tool such as `diff` to compare the baseline Tweet Relevance project to the modified project from this chapter. On a Linux system, for example, the following options for `diff` produce a convenient side-by-side display on a terminal with 237 columns when executed from the root of the source tree:

  ```
  $ diff --recursive --side-by-side --suppress-common-lines --width=237 --
  exclude=*.pyc appa ch04
  ```

  As always, read the `man` page or documentation for the utility you're using to ensure that you're taking advantage of all of the features that you have available to you.
- Modify the example code to implement a parallel payment such that a portion of every payment is donated to a list of prepopulated charities. Why is a parallel payment arguably a more appropriate choice for this situation than a simple or chained payment? (Implementation hint: recall that donating money to a charity with PayPal is as simple as sending money to an email address, so this is essentially as easy as specifying a receiver from a simple HTML control such as a `SELECT` box.)
- Modify the example code to implement an embedded adaptive payment, mimicking the payment flow as implemented with Express Checkout for Digital Goods.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!

CHAPTER 5

Website Payments Pro (Direct Payment)

PayPal's Website Payments Pro product combines the convenience of Express Checkout, an overtly PayPal-oriented checkout, with an additional option called Direct Payment, which allows you to completely embed the user experience for the entire payment process into your site with no mention of PayPal whatsoever required. Even on your customer's credit and debit card statements, there is no mention of PayPal; your company name appears instead. If for whatever reason you'd like to control the entire checkout and provide the most seamless experience possible without any mention of a third party, the Direct Payment portion of Website Payments Pro may be exactly what you've been looking for. Since Express Checkout, the other component of Website Payments Pro, has been covered earlier in the book, getting you up and running with Direct Payment is the primary subject of this chapter.

PayPal official documentation on Website Payments Pro is available online: Website Payments Pro Developer Guide.

Website Payments Pro versus Website Payments Standard

It would be remiss to introduce Website Payments Pro without mentioning that there's also a Website Payments Standard Product, and unfortunately, the difference between the two products is a common source of confusion. In a nutshell, the primary differences between Website Payments Standard and Pro is that Website Payments Standard is completely free, whereas Website Payments Pro involves a small monthly fee, and Website Payments Standard entails a redirect from your site to PayPal (where they enter in credit card information directly at PayPal in a customizable page template) and back, whereas Website Payments Pro provides a completely seamless experience where no redirect is involved.

In short, Website Payments Standard allows your customers to use a credit card to pay without having a PayPal account, it does not cost you anything, and it does not require you to handle sensitive account information directly and incur the potential liability that comes along with it. From the standpoint of the shopper's user experience, the primary distinguishing factor between Standard and Pro is whether or not you want the completely seamless user experience.

PayPal's Standard versus Pro product comparison contains a short video that may be helpful in further clarifying the difference between these two products.

Overview of Direct Payment

In exchange for a nominal monthly fee, the Direct Payment portion of Website Payments Pro allows your customers to pay via credit or debit cards directly on your site. As the seller, this gives you complete control over the buyer's transaction experience without the need for any redirect popup windows, or light boxes or any other friction. Such an arrangement may be appealing in that it provides you with complete control and removes an additional party (PayPal) from the checkout, but do not take lightly the fact that it makes the seller/merchant responsible for maintaining the security of the transaction and that some customers may actually prefer a checkout experience in which PayPal acts as a trusted intermediary instead of providing you with direct access to sensitive account information. Besides, you are actually required by the Website Payments Pro terms of service to use Direct Payment in conjunction with Express Checkout; Direct Payment may not be used as a standalone product.

It is absolutely critical that you provide the Direct Payment checkout experience under an SSL connection, and that you avoid logging or inadvertently storing any sensitive account information associated with a Direct Payment unless it is your explicit intention to do so, implying that you are prepared to safeguard it in accordance with PCI compliance, security best practices, and any applicable laws.

Figure 5-1 shows a typical checkout workflow a user experiences with Direct Payment:

1. The buyer clicks the Checkout button on your website, provides shipping and billing information, and clicks Continue.
2. The buyer reviews the order for accuracy and clicks Pay.
3. Information is handed off to PayPal via the `DoDirectPayment` API operation, the buyer's card is charged, and you are provided with an appropriate response by PayPal.
4. The customer receives an acknowledgment that the order was successfully processed.

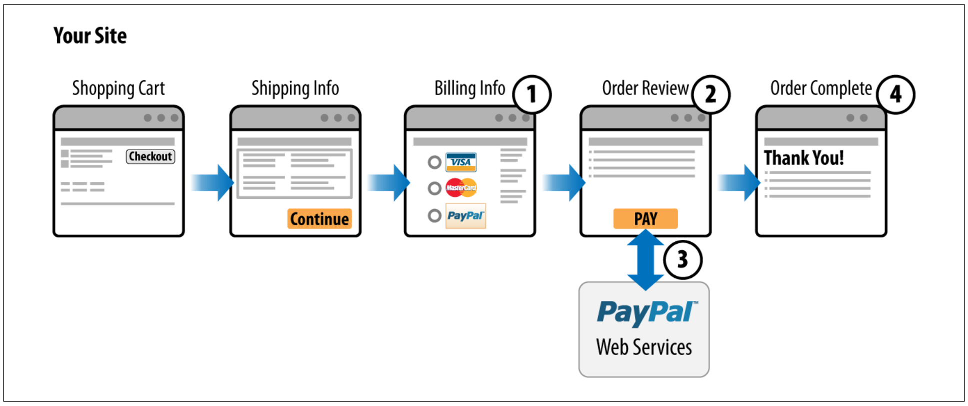


Figure 5-1. PayPal Direct Payment workflow

PayPal Direct Payment API Operations

The PayPal NVP API uses only one method related to Direct Payment: `DoDirectPayment`. This one method initializes the payment and returns the results all in one operation. Table 5-1 outlines the `DoDirectPayment` request fields, and Table 5-2 outlines the method's response fields.

Table 5-1. DoDirectPayment request fields

Field	Description
`METHOD`	Must be `DoDirectPayment` (required).
`PAYMENTACTION`	Indicates how you want to obtain payment: • `Authorization`: This payment is a basic authorization subject to settlement with PayPal Authorization and Capture. • `Sale`: This is the default value, indicating that it is a final sale. Limitation: Up to 13 single-byte characters.
`IPADDRESS`	The IP address of the buyer's browser (required). PayPal records this IP address to detect possible fraud. Limitation: Up to 15 single-byte characters, including periods. Must be an IPv4 address.
`RETURNFMFDETAILS`	Flag that indicates whether you want the results returned by the Fraud Management Filters: • `0`: Do not receive FMF details (default) • `1`: Receive FMF details
`CREDITCARDTYPE`	The type of credit card being used. Allowed values are: • `Visa` • `MasterCard` • `Discover` • `Amex`

Field	Description
	• `Maestro`* • `Solo`* * If using `Maestro` or `Solo`, the `CURRENCYCODE` must be GBP. Additionally, either `START DATE` or `ISSUENUMBER` must be specified. Limitation: Up to 10 single-byte alphabetic characters. For the UK, only `Maestro`, `Solo`, `MasterCard`, `Discover`, and `Visa` are allowed. For Canada, only `MasterCard` and `Visa` are allowed.
`ACCT`	The customer's credit card number. Limitations: Numeric characters only, with no spaces or punctuation. Must conform with the modulo and length required by each card type.
`EXPDATE`	The credit card expiration date, in the format MMYYYY. Limitations: Six single-byte alphanumeric characters, including the leading 0.
`CVV2`	The card verification value, version 2. This field may or may not be required, depending on your merchant account settings. The character length for Visa, MasterCard, and Discover is three digits. The character length for American Express is four digits. To adhere to credit card processing regulations, you cannot store this value after a transaction is complete.
`STARTDATE`	The month and year that a Maestro or Solo card was issued, in MMYYYY format. This value must be six digits, including the leading zero.
`ISSUENUMBER`	The issue number of a Maestro or Solo card. Two numeric digit maximum.
`EMAIL`	The email address of the buyer. Limited to 127 single-byte characters.
`FIRSTNAME`	The buyer's first name (required). Limited to 25 single-byte characters.
`LASTNAME`	The buyer's last name (required). Limited to 25 single-byte characters.
`STREET`	The first street address (required). Limited to 100 single-byte characters.
`STREET2`	The second street address (required). Limited to 100 single-byte characters.
`CITY`	The name of the city (required). Limited to 40 single-byte characters.
`STATE`	The state or province (required). Limited to 40 single-byte characters.
`COUNTRYCODE`	The country code (required). Limited to two single-byte characters.
`ZIP`	The US zip code or another country-specific postal code (required). Limited to 20 single-byte characters.
`SHIPTOPHONENUM`	The phone number. Limited to 20 single-byte characters.
`AMT`	The total cost of the transaction to the customer (required). If the shipping cost and tax charges are known, include them in this value; if not, this value should be the current subtotal of the order. If the transaction includes one or more one-time purchases, this field must be equal to the sum of the purchases. Set this field to `0` if the transaction does not include a one-time purchase, for example, when you set up a billing agreement for a recurring payment that is not charged immediately. Purchase-specific fields will be ignored. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).

Field	Description
CURRENCYCODE	A three-character currency code. The default is USD.
ITEMAMT	Sum of the cost of all items in this order. ITEMAMT is required if you specify L_AMT*n*. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
SHIPPINGAMT	Total shipping cost for this order. If you specify a value for SHIPPINGAMT, you are required to specify a value for ITEMAMT as well. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
HANDLINGAMT	Total handling costs for this order. If you specify a value for HANDLINGAMT, you are required to specify a value for ITEMAMT as well. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
TAXAMT	Sum of the tax for all items in this order. TAXAMT is required if you specify L_TAXAMT*n*. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
DESC	A description of the items the customer is purchasing. Limited to 127 single-byte alphanumeric characters.
CUSTOM	A free-form field for your own use. Limited to 256 single-byte alphanumeric characters.
INVNUM	Your own internal invoice or tracking number. Limited to 127 single-byte alphanumeric characters.
BUTTONSOURCE	An identification code for use by third-party applications to identify transactions. Limited to 32 single-byte alphanumeric characters.
L_NAME *n*	The item name. Limited to 127 single-byte characters.
L_DESC *n*	The item description. Limited to 127 single-byte characters.
L_AMT *n*	The cost of the item. If you specify a value for L_AMT*n*, you must specify a value for ITEMAMT. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
L_NUMBER *n*	The item number. Limited to 127 single-byte characters.
L_QTY *n*	The item quantity. Can be any positive integer.
L_TAXAMT *n*	The item's sales tax. Limitations: Must not exceed $10,000 USD in any currency. No currency symbol. Must have two decimal places, the decimal separator must be a period (.), and the optional thousands separator must be a comma (,).
SHIPTONAME	The person's name associated with the shipping address. Required if using a shipping address. Limited to 32 single-byte characters.
SHIPTOSTREET	The first street address. Required if using a shipping address. Limited to 100 single-byte characters.
SHIPTOSTREET2	The second street address. Limited to 100 single-byte characters.
SHIPTOCITY	The name of the city. Required if using a shipping address. Limited to 40 single-byte characters.
SHIPTOSTATE	The state or province. Required if using a shipping address. Limited to 40 single-byte characters.

Field	Description
SHIPTOZIP	The US zip code or other country-specific postal code. Required if using a US shipping address and might be required for other countries. Limited to 20 single-byte characters.
SHIPTOCOUNTRY	The country code. Required if using a shipping address. Limited to two single-byte characters.
SHIPTOPHONENUM	The phone number. Limited to 20 single-byte characters.

Table 5-2. DoDirectPayment response fields

Field	Description
TRANSACTIONID	The unique transaction ID of the payment. If the `PaymentAction` of the request was `Authorization`, the value of `TransactionID` is your `AuthorizationID` for use with the Authorization and Capture API.
AMT	This value is the amount of the payment you specified in the `DoDirectPayment` request.
AVSCODE	The Address Verification System response code. Limited to one single-byte alphanumeric character.
CVV2MATCH	The results of the CVV2 check by PayPal.
L_FMF*filterID*n	The filter ID, including the filter type (PENDING, REPORT, or DENY), the *filterID*, and the entry number, *n*, starting from 0. *filterID* is one of the following values [AVS stands for Address Verification System]: • 1 = AVS No Match • 2 = AVS Partial Match • 3 = AVS Unavailable/Unsupported • 4 = Card Security Code (CSC) Mismatch • 5 = Maximum Transaction Amount • 6 = Unconfirmed Address • 7 = Country Monitor • 8 = Large Order Number • 9 = Billing/Shipping Address Mismatch • 10 = Risky ZIP Code • 11 = Suspected Freight Forwarder Check • 12 = Total Purchase Price Minimum • 13 = IP Address Velocity • 14 = Risky Email Address Domain Check • 15 = Risky Bank Identification Number (BIN) Check • 16 = Risky IP address Range • 17 = PayPal Fraud Model
L_FMF*filterNAME*n	The filter name, including the filter type, (PENDING, REPORT, or DENY), the *filterNAME*, and the entry number, *n*, starting from 0.

Implementing DoDirectPayment

In this section, we'll learn how to implement an abstract `DoDirectPayment` API request and validate it by using `curl` before transitioning into a GAE implementation. In the next section, we'll integrate `DoDirectPayment` into Tweet Relevance.

DoDirectPayment API Operation

To implement a Direct Payment transaction, you need to invoke the `DoDirectPayment` API and provide information to identify the buyer's credit or debit card and the amount of the payment. Setting up a minimal transaction is accomplished through the following steps in which you construct a URL request to the PayPal API endpoint for `DoDirectPayment`:

1. Specify the amount of the transaction, including the currency if it is not in US dollars. You should specify the total amount of the transaction if it is known; otherwise, specify the subtotal:

   `AMT=` *`amount`*

   `CURRENCYCODE=` *`currencyID`*

2. Specify the payment action. It is best practice to *explicitly* specify the payment action as one of the following values, even though the default value is `Sale`:

   `PAYMENTACTION=Sale`

   `PAYMENTACTION=Authorization`

3. Specify the IP address of the buyer's computer:

   `IPADDRESS=` *`xxx.xxx.xxx.xxx`*

4. Specify information about the card being used. You must specify the type of card as well as the account number:

   `CREDITCARDTYPE=Visa`

   `ACCT=` *`1234567891011123`*

   The type of credit/debit card being used, the card issuer, and the Payment Receiving Preferences setting on your PayPal Profile might require that you set the following fields as well:

   `EXPDATE=` *`012013`*

   `CVV2=` *`123`*

5. Specify information about the card holder. You must provide the first and last name of the card holder, as well as the billing address associated with the card:

   `FIRSTNAME=` *`John`*

   `LASTNAME=` *`Doe`*

   `STREET=` *`1313 Mockingbird Lane`*

CITY= *Franklin*

STATE= *TN*

ZIP= *37064*

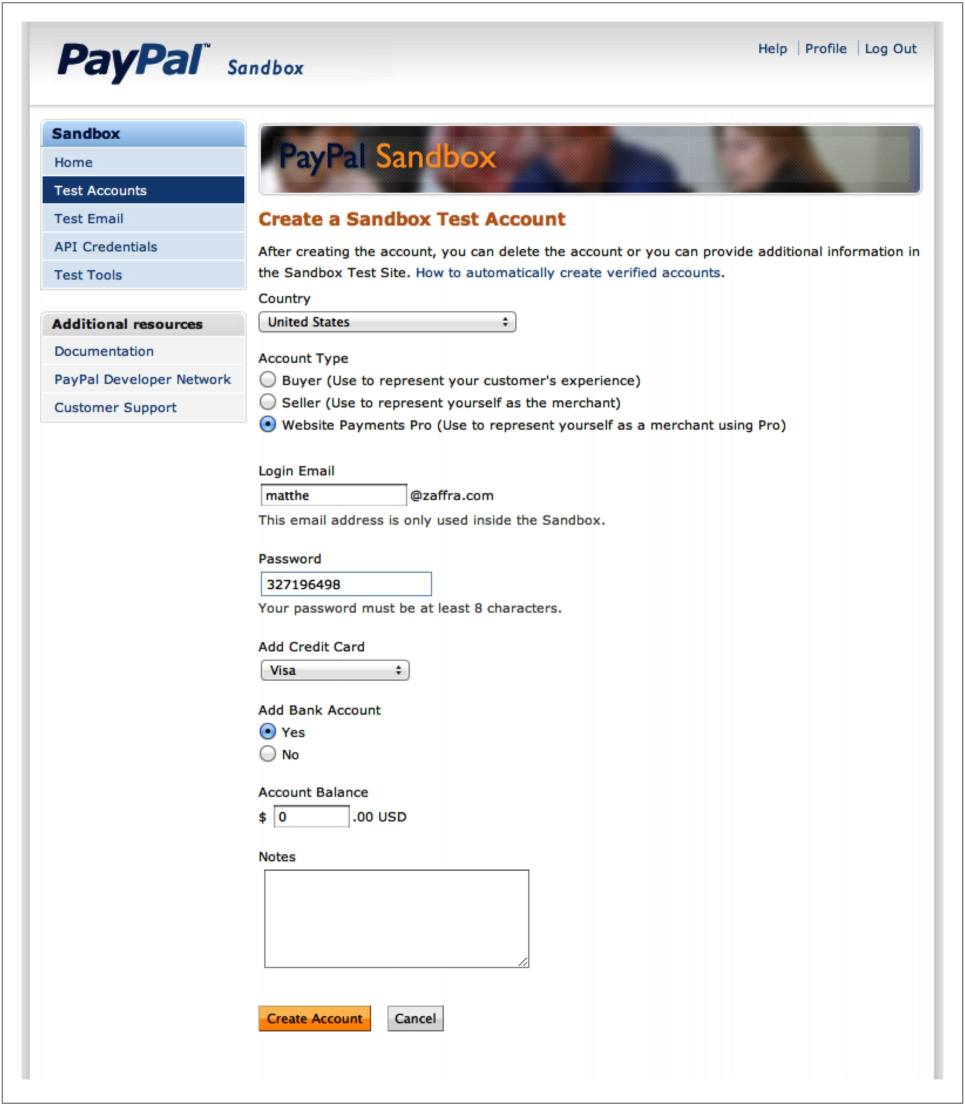
COUNTRYCODE= *US*

A minimal Python implementation of a `DoDirectPayment` transaction using GAE is coming up shortly. First, however, it may be helpful to execute a transaction using a shell script using `curl` to ensure that your buyer and seller configuration is setup correctly. In order to use `DoDirectPayment` as part of Website Payments Pro, you may need to create a business account in the developer sandbox environment and explicitly indicate that the account is a merchant account for Website Payments Pro, as shown in Figure 5-2. The 3-Token credentials associated with this account should allow you to process `DoDirectPayment` transactions, whereas credentials associated with a Seller account should not.

Chances are pretty good that you already have one or more personal accounts configured in the developer sandbox that you've already been using as a buyer or receiver of a payment. Ensure that you are using the credit card information associated with one of these accounts, because the `DoDirectPayment` API will validate the `DoDirectPayment` transaction using this faux credit card number and expiration date. [A live `DoDirect Payment` transaction may validate all other fields and provide back any of many possible error codes for other validation issues that you should be prepared to handle.] An example shell script that demonstrates the `DoDirectPayment` using the guidance provided in this section is shown in Example 5-1. If you are comfortable working with shell scripts in a Linux environment or are able to reconfigure the script to execute on Windows, it is highly recommended that you replace the variables at the top of the script with your own merchant account 3-Token credentials and buyer account credit card information and receive a successful response before proceeding. If you are unable to run the script, don't worry—a Python example with GAE is coming up shortly. The point of including a shell script here is to show that the actual execution of a `DoDirect Payment` is truly just a single HTTP request.

Example 5-1. DoDirectPayment.sh—a minimal Bash script illustrating the use of the DoDirect Payment API

```
#!/bin/bash

# Ensure that the 3-Token credentials are with a business account that has explicitly
# been configured for Website Payments Pro at http://developer.paypal.com

USER="XXX"
PWD="XXX"
SIGNATURE="XXX"

# Ensure that the credit card used is a real credit card number from an account
# that has been configured at http://developer.paypal.com
# The sandbox environment does require that the credit card number and expiry
```



Create a Sandbox Test Account

After creating the account, you can delete the account or you can provide additional information in the Sandbox Test Site. How to automatically create verified accounts.

Country: United States

Account Type
- Buyer (Use to represent your customer's experience)
- Seller (Use to represent yourself as the merchant)
- [✓ Website Payments Pro (Use to represent yourself as a merchant using Pro)]

Login Email: matthe @zaffra.com
This email address is only used inside the Sandbox.

Password: 327196498
Your password must be at least 8 characters.

Add Credit Card: Visa

Add Bank Account
[✓ Yes]
No

Account Balance
$ 0 .00 USD

Notes

Figure 5-2. Creating a merchant account in the developer sandbox for Website Payments Pro

```
# be valid

ACCT="XXX"
EXPDATE="XXX"

RESULT=$(curl -s \
https://api-3t.sandbox.paypal.com/nvp -d "VERSION=82.0\
&USER=$USER\
&PWD=$PWD\
&SIGNATURE=$SIGNATURE\
&METHOD=DoDirectPayment\
&PAYMENTACTION=Sale\
&IPADDRESS=192.168.0.1\
&AMT=8.88\
&CREDITCARDTYPE=Visa\
&ACCT=$ACCT\
&EXPDATE=$EXPDATE\
&CVV2=123\
&FIRSTNAME=John\
&LASTNAME=Smith\
&STREET=1000 Elm St.\
&CITY=Franklin\
&STATE=TN\
&ZIP=37064\
&COUNTRYCODE=US"\
;)

echo $RESULT;
```

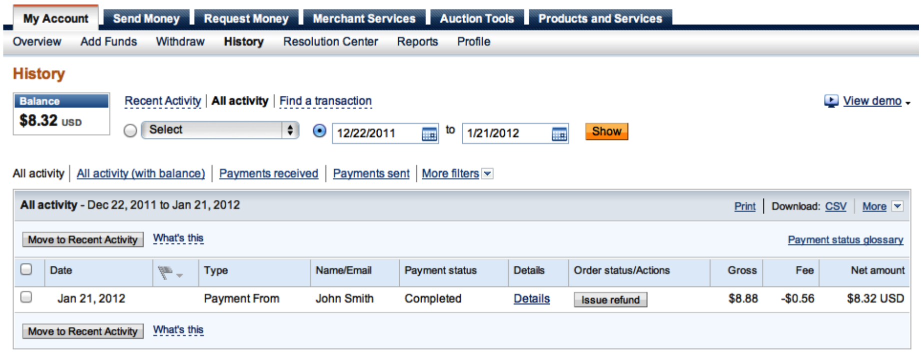
A successful response from the script should look something like the following:

```
TIMESTAMP=2012%2d01%2d22T02%3a14%3a59Z
&CORRELATIONID=46d8df6362e04
&ACK=Success
&VERSION=82%2e0
&BUILD=2278658
&AMT=8%2e88
&CURRENCYCODE=USD
&AVSCODE=X
&CVV2MATCH=M
&TRANSACTIONID=0BD118857L408034Y
```

Your business logic should be prepared to fully parse the results and be prepared to take all necessary actions for contingent situations such as aberrations with `AVSCODE` or `CVV2MATCH`, even if the `ACK` field is `Success`. For example, you should be prepared to handle the case when `AVSCODE` isn't `X`, which indicates an "exact match" according to the AVS and CVV2 Response Codes documentation. The official documentation on `DoDirectPayment` also contains information on how to use PayPal's fraud management filters to help you identify potentially fraudulent transactions—another important consideration you should be prepared to tackle with a `DoDirectPayment` implementation.

Even though the PayPal Sandbox environment doesn't appear to validate fields pertaining to `AVSCODE` and `CVV2MATCH`, such as street address, zip code, or the three-digit code on the backs of credit cards, it's critical that you think through and simulate testing scenarios before moving into production with `DoDirectPayment`.

Like all other transactions in the Sandbox environment, you are able to log into the merchant account and view the details of the transaction in the account history. Figure 5-3 shows what the merchant account history might look like after a successful transaction.



Date	Type	Name/Email	Payment status	Details	Order status/Actions	Gross	Fee	Net amount
Jan 21, 2012	Payment From	John Smith	Completed	Details	Issue refund	$8.88	-$0.56	$8.32 USD

Figure 5-3. Creating a merchant account in the developer sandbox for Website Payments Pro

Implementing DoDirectPayment with GAE

Unlike the other PayPal products that have been demonstrated thus far, `DoDirectPayment` requires only a single request and response. In terms of the buyer's user experience, PayPal isn't playing an intermediary role and you are handling the account information yourself, so there's no need for a more roundabout approval process involving redirects. Thus, the implementation details for completing a `DoDirectPayment` transaction are considerably simpler, as shown in Example 5-2.

Example 5-2. main.py—a minimal DoDirectPayment example using GAE

```
"""
A minimal GAE application that makes an API request to PayPal
and parses the result. Fill in your own 3 Token Credentials
from your sandbox account
"""

from google.appengine.ext import webapp
from google.appengine.ext.webapp import util
from google.appengine.api import urlfetch

import urllib
import cgi

# Replace these values with your own 3-Token credentials from a sandbox
# merchant account that was configured for Website Payments Pro and a faux
# credit card number and expiry from a "personal" buyer account

user_id = "XXX"
password = "XXX"
signature = "XXX"
```

```
credit_card_number = "XXX"
credit_card_expiry ="XXX"

class MainHandler(webapp.RequestHandler):
    def get(self):

        # Sandbox NVP API endpoint

        sandbox_api_url = 'https://api-3t.sandbox.paypal.com/nvp'

        nvp_params = {
            # 3 Token Credentials

            'USER' : user_id,
            'PWD' : password,
            'SIGNATURE' : signature,

            # API Version and Operation

            'METHOD' : 'DoDirectPayment',
            'VERSION' : '82.0',

            # API specifics for DoDirectPayment
            'PAYMENTACTION' : 'Sale',
            'IPADDRESS' : '192.168.0.1',
            'AMT' : '8.88',
            'CREDITCARDTYPE' : 'Visa',
            'ACCT' : credit_card_number,
            'EXPDATE' : credit_card_expiry,
            'CVV2' : '123',
            'FIRSTNAME' : 'Billy',
            'LASTNAME' : 'Jenkins',
            'STREET' : '1000 Elm St.',
            'CITY' : 'Franklin',
            'STATE' : 'TN',
            'ZIP' : '37064',
            'COUNTRYCODE' : 'US'
        }

        # Make a secure request and pass in nvp_params as a POST payload

        result = urlfetch.fetch(
                    sandbox_api_url,
                    payload = urllib.urlencode(nvp_params),
                    method=urlfetch.POST,
                    deadline=10, # seconds
                    validate_certificate=True
                 )

        if result.status_code == 200: # OK

            decoded_url = cgi.parse_qs(result.content)

            for (k,v) in decoded_url.items():
                self.response.out.write('<pre>%s=%s</pre>' % (k,v[0],))
        else:

            self.response.out.write('Could note fetch %s (%i)' %
                    (url, result.status_code,))

def main():
    application = webapp.WSGIApplication([('/', MainHandler)],
                                         debug=True)
    util.run_wsgi_app(application)

if __name__ == '__main__':
    main()
```

There should be little explanation that's required for understanding this example if you've read previous chapters. A single API request is passed in with all of the appropriate parameters, and a response is returned with the standard fields from Table 5-2, indicating success or failure along with a few other pertinent details. If you haven't already, take the time to run the example to ensure that you have successfully configured your sandbox merchant account to be enabled for Website Payments Pro and are using valid credit card information from a sandbox personal account. The next section integrates `DoDirectPayment` into Tweet Relevance.

Integrating DoDirectPayment and Tweet Relevance

Like every other chapter in this book, let's take the Tweet Relevance sample code from Appendix A and use the PayPal product at hand, `DoDirectPayment` in this case, to implement a payment experience. Although an integration with `DoDirectPayment` as part of a Website Payments Pro integration normally requires additional integration with Express Checkout according to PayPal's terms of service, we'll focus solely on integrating `DoDirectPayment` in this chapter to maintain maximal focus. (Recall that Express Checkout is discussed at length in Chapters 2 and 3.) A recommended exercise for the seriously interested reader, as presented in the final section of this chapter, is to integrate Express Checkout into this chapter's sample project.

It may be helpful to review "Implementing a Checkout Experience for Tweet Relevance" on page 32 to better understand some of the various payment mechanisms that could be viable for a service like Tweet Relevance if you have not done so already. The remainder of this chapter assumes familiarity with the options as presented in that section and implements the "subscription model," which was covered in detail in Chapter 2 and used again in Chapter 4.

The first step to integrating `DoDirectPayment` or any other payment mechanism is to map a URL into the main application as a means of handling a payment experience. Since integrating `DoDirectPayment` requires only a single call to PayPal, the addition of a /do_direct_payment URL that's serviced by a PaymentHandler class is the only API-level addition to the application that is necessary. A basic template that we'll add to collect payment information from the user once their free trial of the service expires will pass the information to /do_direct_payment, which is the entry point for the payment process. The following list itemizes the key changes to the project for integrating DoDirectPayment in a file-by-file fashion.

main.py

Example 5-3 illustrates the entry point into the web application. Note that there's only one PaymentHandler URL.

Example 5-3. main.py

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp import util

# Logic for implementing DoDirectPayment

from handlers.PaymentHandler import PaymentHandler

# Logic for the app itself

from handlers.AppHandler import AppHandler

# Logic for interacting with Twitter's API and serving up data, etc.

def main():

  application = webapp.WSGIApplication([

                                         # PaymentHandler URLs

                                         ('/(do_direct_payment)', PaymentHandler),

                                         # AppHandler URLs

                                         ('/(app)', AppHandler),
                                         ('/(data)', AppHandler),
                                         ('/(login)', AppHandler),
                                         ('/', AppHandler)
                                        ],

                                        debug=True)
  util.run_wsgi_app(application)

if __name__ == '__main__':
  main()
```

handlers/PaymentHandler.py

A standard form that collects payment information, as shown in Figure 5-4, passes this information through to /do_direct_payment as a POST request, which is serviced by PaymentHandler, as shown in Example 5-5. In a nutshell, PaymentHandler validates the payment information, passes it through to `DoDirectPayment`, and credits the user's account if the payment action was successful. Otherwise, it displays an error message. The trivial `Product` class that's referenced by `PaymentHandler` is shown in Example 5-4.

Example 5-4. Product.py

```
# The Product class provides product details.
# A more flexible product line could be managed in a database

class Product(object):

  @staticmethod
  def getProduct():

    return {'price' : 9.99, 'quantity' : 30, 'units' : 'days'}
```

Example 5-5. handlers/PaymentHandler.py

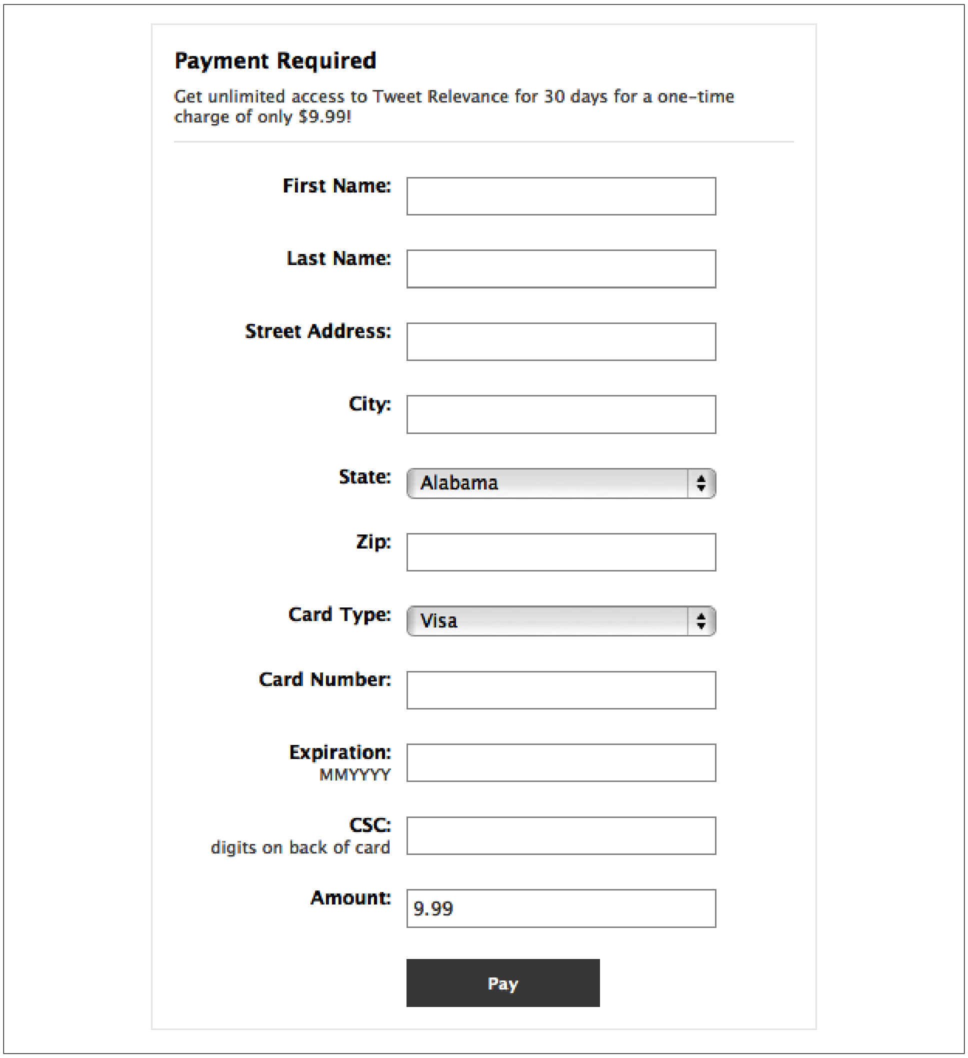
```
# PaymentHandler provides logic for interacting wtih PayPal's Website Payments Pro
product

import os
import cgi

from google.appengine.ext import webapp
from google.appengine.api import memcache
from google.appengine.ext.webapp import template
import logging

from paypal.products import DirectPayment as DP
from Product import Product
from handlers.AppHandler import AppHandler


class PaymentHandler(webapp.RequestHandler):

  def post(self, mode=""):

    if mode == "do_direct_payment":

     # To be on the safe side, filter through a pre-defined list of fields
     # to pass through to DoDirectPayment. i.e. prevent the client from
     # potentially overriding IPADDRESS, AMT, etc.

      valid_fields = [
          'FIRSTNAME',
          'LASTNAME',
          'STREET',
          'CITY',
          'STATE',
          'ZIP',
          'COUNTRYCODE',
          'CREDITCARDTYPE',
          'ACCT',
          'EXPDATE',
          'CVV2',
      ]

      product = Product.getProduct()

      nvp_params = {'AMT' : str(product['price']), 'IPADDRESS' :
self.request.remote_addr}

      for field in valid_fields:
        nvp_params[field] = self.request.get(field)

      response = DP.do_direct_payment(nvp_params)

      if response.status_code != 200:
        logging.error("Failure for DoDirectPayment")

        template_values = {
          'title' : 'Error',
          'operation' : 'DoDirectPayment'
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unknown_error.html')
        return self.response.out.write(template.render(path, template_values))

      # Ensure that the payment was successful

      parsed_qs = cgi.parse_qs(response.content)

      if parsed_qs['ACK'][0] != 'Success':
        logging.error("Unsuccessful DoDirectPayment")

        template_values = {
          'title' : 'Error',
          'details' : parsed_qs['L_LONGMESSAGE0'][0]
        }

        path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'unsuccessful_payment.html')
        return self.response.out.write(template.render(path, template_values))


      # Credit the user's account

      user_info = memcache.get(self.request.get("sid"))
      twitter_username = user_info['username']
      product = Product.getProduct()

      AppHandler.creditUserAccount(twitter_username, product['quantity'])

      template_values = {
        'title' : 'Successful Payment',
        'quantity' : product['quantity'],
        'units' : product['units']
      }

      path = os.path.join(os.path.dirname(__file__), '..', 'templates',
'successful_payment.html')
      self.response.out.write(template.render(path, template_values))

    else:
      logging.error("Unknown mode for POST request!")
```

products/paypal.py

The final piece of substance to completing the discussion on a `DoDirectPayment` integration involves the `DirectPayment` class that's referenced in `PaymentHandler`. Basically, this class is just a thin abstraction around the `DoDirectPayment` API call and follows along with the same pattern used for the `paypal` package in earlier chapters. In short, it combines the 3-Token credentials with any other keyword parameters passed into the `do_direct_payment` method and executes a `DoDirectPayment` API call.

Example 5-6. paypal/products.py

```
from google.appengine.api import urlfetch

import urllib
import cgi

import paypal_config

class DirectPayment(object):

    @staticmethod
    def _api_call(nvp_params):

      params = nvp_params.copy()      # copy to avoid mutating nvp_params with update()
       params.update(paypal_config.nvp_params) # update with 3 token credentials and
api version

        response = urlfetch.fetch(
                    paypal_config.sandbox_api_url,
                    payload=urllib.urlencode(params),
                    method=urlfetch.POST,
                    validate_certificate=True,
                    deadline=10 # seconds
                   )

        if response.status_code != 200:
            decoded_url = cgi.parse_qs(result.content)

            for (k,v) in decoded_url.items():
                logging.error('%s=%s' % (k,v[0],))

            raise Exception(str(response.status_code))

        return response
    @staticmethod
    def do_direct_payment(nvp_params):
        nvp_params.update(METHOD='DoDirectPayment')
        nvp_params.update(PAYMENTACTION='Sale')
        return DirectPayment._api_call(nvp_params)
```



Payment Required

Get unlimited access to Tweet Relevance for 30 days for a one-time charge of only $9.99!

First Name:	
Last Name:	
Street Address:	
City:	
State:	Alabama
Zip:	
Card Type:	Visa
Card Number:	
Expiration: MMYYYY	
CSC: digits on back of card	
Amount:	9.99

Figure 5-4. An austere form for collecting payment information that's passed through to / do_direct_payment. (See templates/checkout.html in the Tweet Relevance project file for this chapter.)

Prior chapters, which covered products explicitly involving PayPal as an intermediating party in the user experience of the application, required more than a single API call to PayPal due to very the nature of setting up a payment, redirecting the customer to PayPal (to avoid having you directly handle sensitive account information), and ensuring the payment was processed before crediting a user's account. Because `DoDirectPayment` gives you the ability handle payment account information directly, some of the implementation details are a bit simpler because of the streamlined user experience. Just remember that with the power to handle sensitive account information directly comes great responsibility and (potentially) great liability as it relates to maintaining PCI compliance and safeguarding financial records.

Recommended Exercises

- Integrate the Express Checkout code from Chapter 2 with the project code from this chapter to "fully implement" Website Payments Pro as required by PayPal's terms of service. Hint: consider maintaining separate payment handling classes for each of these products and making just a few minor changes to *templates/checkout.html*.
- Use a tool such as `diff` to compare the baseline Tweet Relevance project to the modified project from this chapter. On a Linux system, for example, the following options for `diff` produce a convenient side-by-side display on a terminal with 237 columns when executed from the root of the source tree:

  ```
  $ diff --recursive --side-by-side --suppress-common-lines --width=237 --
  exclude=*.pyc appa ch04
  ```
- Try implementing an "authorization and capture" as described in the Website Payments Pro Integration Guide to implement an extension to Tweet Relevance in which a paying customer can gift a 30-day subscription to a friend. Authorize the card at the time the paying customer gifts the subscription, but don't capture the funds until the recipient of the gift subscription logs in for the first time.
- Read the Wikipedia article on PCI Compliance.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!

CHAPTER 6

Instant Payment Notifications (IPNs)

Instant Payment Notifications (IPNs) are messages that PayPal sends you regarding a transaction status and can serve a variety of useful purposes. This brief chapter introduces IPNs, covers some of the common use cases for when IPNs can be helpful, and ultimately, teaches you how to use them to perform a follow-up action after a purchase as part of a Tweet Relevance integration.

PayPal's official documentation on IPNs is available online: Instant Payment Notification Guide.

Overview of IPNs

IPNs notify you when an event occurs that affects a transaction. Typically, these events represent various kinds of payments but could also represent authorizations, Fraud Management Filter actions, refunds, disputes, chargebacks, the status of an eCheck payment, etc. More specifically, IPNs are commonly employed regarding:

- Instant payments, including Express Checkout, Adaptive Payments, and direct credit card payments, and authorizations, which indicate a sale whose payment has not yet been collected
- eCheck payments and associated status, such as pending, completed, or denied, and payments pending for other reasons, such as those being reviewed for potential fraud
- Recurring payment and subscription actions
- Chargebacks, disputes, reversals, and refunds associated with a transaction

See "Instant Payments versus eChecks" on page 31 for a brief overview of eChecks versus instant payments.

In many cases, the action that causes the event, such as a payment, occurs on your website; however, your website is not the only source of events. In many cases, events can be generated by Website Payment Standard buttons (not covered in this book) or various PayPal APIs such as `DoExpressCheckoutPayment`. You can detect and process IPN messages with a listener script that waits for messages and passes them to various backend or administrative processes that respond to them. The actions to take when your listener is notified of an event are specific to your needs. Examples of the kinds of actions you might take when your listener receives an IPN message include:

- Trigger order fulfillment or enable media downloads when an eCheck clears or a payment is made
- Update your list of customers
- Update accounting records
- Send follow-up emails regarding the transaction

You are typically notified of events by email as well, but the IPN message service enables you to automate a response to events using a well-formed API. Additionally, IPNs are asynchronous, which means that messages are not synchronized with actions on your website. Thus, listening for an IPN message does not increase the time it takes to complete a transaction on your website.

The IPN message service does not assume that all messages will be received by your listener in a timely manner. Because the Internet is not 100% reliable, messages can become lost or delayed. To handle the possibility of transmission and receipt delays or failures, the IPN message service implements an algorithm that resends messages at various intervals until you acknowledge that the message has successfully been received. The delay between each successive resend attempt increases, and the IPN message service will retry for up to four days after the original message attempt.

Because messages can be delivered at any time, your listener must always be available to receive and process messages; however, the retry mechanism also handles the possibility that your listener could become swamped or stop responding. The IPN message service should not be considered a real-time service, and your checkout flow should not wait on an IPN message before it is allowed to complete. If your website waits for an IPN message, checkout processing may be delayed due to system load and become even more complicated because of the possibility of retries.

IPN Protocol and Architecture

IPN is designed to be secure, reliable, and asynchronous. To meet these requirements, the protocol requires you to acknowledge receipt of IPN messages. The IPN service provides a retry mechanism to handle cases in which a message is not acknowledged, for example, when a transmission or receipt failure occurs. When you enable IPN, PayPal sends messages to the IPN listener at the URL you specify in your merchant account's profile, although you can override the URL to associate other IPN listeners with specific transactions. In this case, you specify the listener's URL when you set up a PayPal API operation. The IPN protocol consists of three steps:

1. PayPal sends your IPN listener a message that notifies you of the event.
2. Your listener sends the complete, unaltered message back to PayPal, prepended with the request parameters `cmd=_notify-validate`.
3. PayPal sends a single word back, `VERIFIED` or `INVALID`, indicating whether or not the IPN originated with PayPal.

Your listener must respond to each message, whether or not you intend to do anything with it, because PayPal assumes that the message was not received and resends the message for up to four days if you do not respond. The resend algorithm increases the interval between each successive retry, which can lead to an interesting edge case: Pay-Pal may resend the IPN message while you are sending back the original message. In this case, you should send your response again to cover the possibility that PayPal did not actually receive your response the first time. You should also ensure that you do not process the transaction associated with the message twice, which is usually handled by persisting all transaction IDs that you have processed.

Once PayPal verifies the IPN, there are additional checks that your listener should take before it performs any follow-up actions:

- Verify that you are the intended recipient of the IPN message by checking the email address in the message; this handles a situation where another merchant could accidentally or intentionally attempt to use your listener. It's an unlikely but possible circumstance that you should be prepared to handle.
- Avoid duplicate IPN messages. Check that you have not already processed the transaction identified by the transaction ID returned in the IPN message. You may need to store transaction IDs returned by IPN messages in a file or database so that you can check for duplicates. If the transaction ID sent by PayPal is a duplicate, you should *not* process it again.
- Because IPN messages can be sent at various stages in a transaction's progress, make sure that the transaction's payment status is "completed" before enabling shipment of merchandise or allowing the download of digital media. For example, eCheck statuses may initially be "pending" and potentially even end up resulting in a "denied" status.

- Verify that the payment amount actually matches what you intend to charge. Although not technically an IPN issue, if you do not encrypt communications, it is possible for a malicious party to capture the original transmission and change the price. Without this check, you could accept a lesser payment than expected without even realizing it.

PayPal generates an IPN message when you invoke certain API operations such as `DoExpressCheckoutPayment` or `DoDirectPayment` during checkout, or an Adaptive Payments operation such as `Pay`, `Preapproval`, or `ExecutePayment`. Figure 6-1 shows both the web flow and the IPN message authentication protocol.

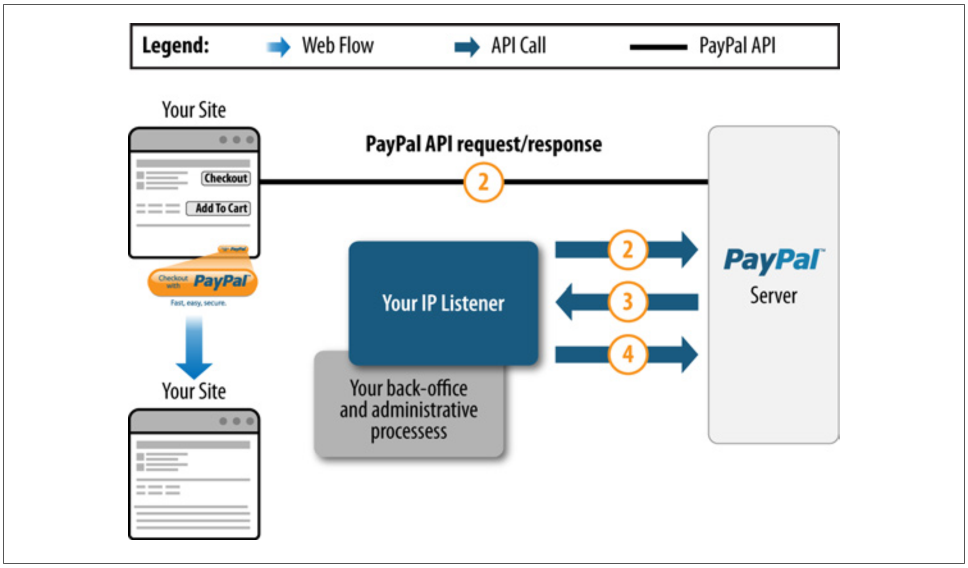


Figure 6-1. Typical IPN flow

The numbers in the diagram correspond to the following steps:

1. The API operation initiates a payment on PayPal.
2. PayPal sends your IPN listener a message that notifies you of the event.
3. Your listener sends the complete, unaltered message back to PayPal, prepending the request parameters with `cmd=_notify-validate`. It is absolutely critical that the message contain the same fields in the same order and be encoded in the same way as the original message.
4. PayPal sends a single word back, which is either `VERIFIED` if the message originated with PayPal or `INVALID` if there is any discrepancy with what was originally sent.

Your IPN listener must implement the IPN authentication protocol (steps 2, 3, and 4 in this diagram). After successfully completing the protocol, your back-office or administrative process vets the contents of the message and responds appropriately. For example, if the payment status for the transaction is "Completed," your system can print a packing list or email a password to your customer for downloading digital media.

PayPal recommends that you follow a specific checklist in handling IPNs:

1. Wait for an HTTP POST from PayPal.
2. Create a request that contains exactly the same IPN variables and values in the same order, preceded with the request parameters `cmd=_notify-validate`.
3. Post back the request to PayPal
4. Wait for a response from PayPal, which is either `VERIFIED` or `INVALID`.
5. Verify that the response status is 200.
6. If the response is `VERIFIED`, perform the following checks:
   - Confirm that the payment status is "Completed." PayPal sends IPN messages for pending and denied payments as well; do not ship until the payment has cleared.
   - Use the transaction ID to verify that the transaction has not already been processed, which prevents duplicate transactions from being processed. Typically, you store transaction IDs in a database so that you know you are only processing unique transactions.
   - Validate that the receiver's email address is registered to you. This check provides additional protection against fraud.
   - Verify that the price, item description, and so on, match the transaction on your website. This check provides additional protection against fraud.
7. If the verified response passes the checks, take action based on the value of the `txn_type` variable if it exists; otherwise, take action based on the value of the `reason_code` variable.
8. If the response is `INVALID` or the response code is not 200, save the message for further investigation.

Integrating IPNs Into Tweet Relevance

While the previous sections regurgitated much of the introductory content in the Instant Payment Notification Guide and created a necessary foundation, it's all theory until rooted in some sample project code. This section creates a GAE project that implements an IPN listener and teaches you how to use it to perform post-processing actions with the Tweet Relevance sample project from Chapter 2, which involved an Express Checkout integration. The specific post-processing action that we'll perform is to send a follow-up email to the purchaser of a Tweet Relevance subscription after verifying that the IPN originated with PayPal. Although conceptually simple, this approach provides a realistic yet isolated view of what an IPN listener implemented with

GAE might look like and hopefully will serve you well as a jumping off point—not to mention that it demonstrates how to send mail through GAE.

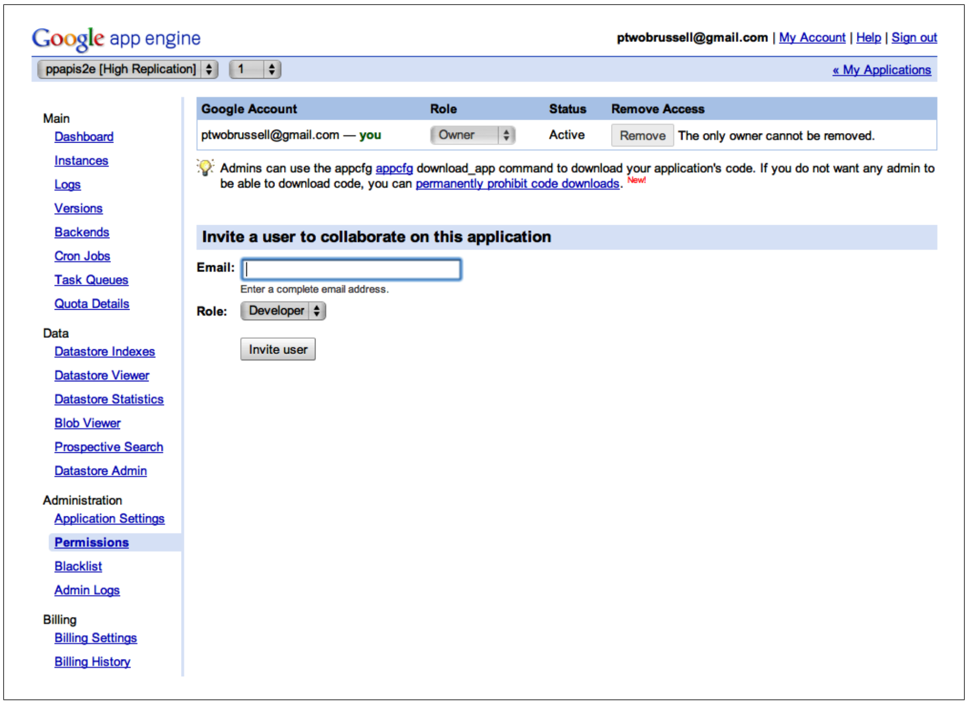
Another obvious option for employing IPNs as part of a Tweet Relevance payment scenario is to use them to take a specific action when an eCheck status is successfully resolved. A recommended exercise for this chapter suggests a scenario involving IPNs and eChecks.

An austere template for interacting with IPNs and performing a custom action based on information provided by the IPN is given in Example 6-1, which shows how you could use an IPN to send a follow-up email message to the payer about her purchase. It does not persist information such as the transaction ID or perform other important implementation details, such as customizing the message based upon whether the payment status is `Completed`. The only change that should be necessary in order to deploy the code to your live GAE environment on AppSpot is specifying an email address for the `gae_email_sender` that is registered with your GAE account, as specified in the Administration/Permissions section as shown in Figure 6-2.

Example 6-1. main.py—a basic template for handling an IPN from PayPal and sending a mail message in response to it as a custom action

```
# Use the IPN Simulator from the "Test Tools" at developer.paypal.com to
# send this app an IPN. This app must be deployed to the live GAE environment
# for the PayPal IPN Simulator to address it. When debugging, it's a necessity
# to use logging messages and view them from the GAE Dashboard's Logs pane.

# You can also login to a Sandbox Merchant account and set the IPN URL for
# this app under the Merchant Profile tab.

from google.appengine.ext import webapp
from google.appengine.ext.webapp import util
from google.appengine.api import urlfetch
from google.appengine.api import mail

import cgi

import logging

# In production, gae_email_sender must be an authorized account that's been
# added to the GAE Dashboard under Administration/Permissions or else you
# won't be able to send mail. You can use the default owner of the account
# or add/verify additional addresses

gae_email_sender = "XXX"
ipn_sandbox_url = "https://www.sandbox.paypal.com/cgi-bin/webscr"

class IPNHandler(webapp.RequestHandler):

    @staticmethod
    def sendMail(first_name, last_name, email, debug_msg=None, bcc=None):
        message = mail.EmailMessage(sender="Customer Support <%s>" % (gae_email_sender,),
                                    subject="Your recent purchase")

        message.to = "%s %s <%s>" % (first_name, last_name, email,)

        message.body = """Dear %s:

Thank you so much for your recent purchase.

Please let us know if you have any questions.

Regards,
Customer Support""" % (first_name,)

        if debug_msg:
            message.body = message.body + '\n' + '*'*20 + '\n' + debug_msg + '\n' + '*'*20

        if bcc:
            message.bcc = bcc

        message.send()


    def post(self, mode=""):

        # PayPal posts to /ipn to send this application an IPN

        if mode == "ipn":

            logging.info(self.request.body)

            # To ensure that it was really PayPal that sent the IPN, we post it back
            # with a preamble and then verify that we get back VERIFIED and a 200 response

            result = urlfetch.fetch(
                        ipn_sandbox_url,
                        payload = "cmd=_notify-validate&" + self.request.body,
                        method=urlfetch.POST,
                        validate_certificate=True
                     )

            logging.info(result.content)

            if result.status_code == 200 and result.content == 'VERIFIED': # OK

                # See pages 19-20 of the Instant Payment Notification Guide at
                # https://cms.paypal.com/cms_content/US/en_US/files/developer/IPNGuide.pdf
                # for various actions that should be taken based on the IPN values.

                ipn_values = cgi.parse_qs(self.request.body)
                debug_msg = '\n'.join(["%s=%s" % (k,'&'.join(v)) for (k,v) in
ipn_values.items()])

               # The Sandbox users don't have real mailboxes, so bcc the GAE email sender
as a way to
```

```
                # debug during development
                self.sendMail(ipn_values['first_name'][0], ipn_values['last_name'][0],
ipn_values['payer_email'],
                              debug_msg=debug_msg, bcc=gae_email_sender)
            else:
                logging.error('Could not fetch %s (%i)' % (url, result.status_code,))

        else:
            logging.error("Unknown mode for POST request!")

def main():
    application = webapp.WSGIApplication([('/', IPNHandler),
                                          ('/(ipn)', IPNHandler)],
                                         debug=True)
    util.run_wsgi_app(application)

if __name__ == '__main__':
    main()
```



Google Account	Role	Status	Remove Access
ptwobrussell@gmail.com — you	Owner	Active	Remove The only owner cannot be removed.

Admins can use the appcfg appcfg download_app command to download your application's code. If you do not want any admin to be able to download code, you can permanently prohibit code downloads. New!

Invite a user to collaborate on this application

Email:

Enter a complete email address.

Role: Developer

Invite user

Figure 6-2. Log in to your merchant account and select "Instant Payment Notification preferences" under the Profile tab in order to add a Notification URL

In short, the sample code accepts a POST request, prepends the mandatory `cmd=_notify-validate` preamble to its body, and sends it back to PayPal. Assuming that PayPal sends back VERIFIED and the response code is 200 (OK), it sends an email to the `payer_email` that's provided in the original IPN message and appends the full contents of the IPN so that you can easily view it without having to log in to the GAE console to view the server logs. Because Sandbox accounts don't have real email addresses, however, it BCCs the `gae_email_sender` to ensure that the email actually arrives at a real address so that you know it's working properly. In production, you'd remove the `bcc` and `debug` parameters.

Take a moment to study the sample code, and then deploy it to AppSpot after you've updated it with a valid email address that is registered with your GAE account. Recall that the IPN handler must be addressable on the Web by PayPal (which cannot be a *http://localhost* address), so you must actually deploy your GAE project code to AppSpot in order to test your IPN listener.

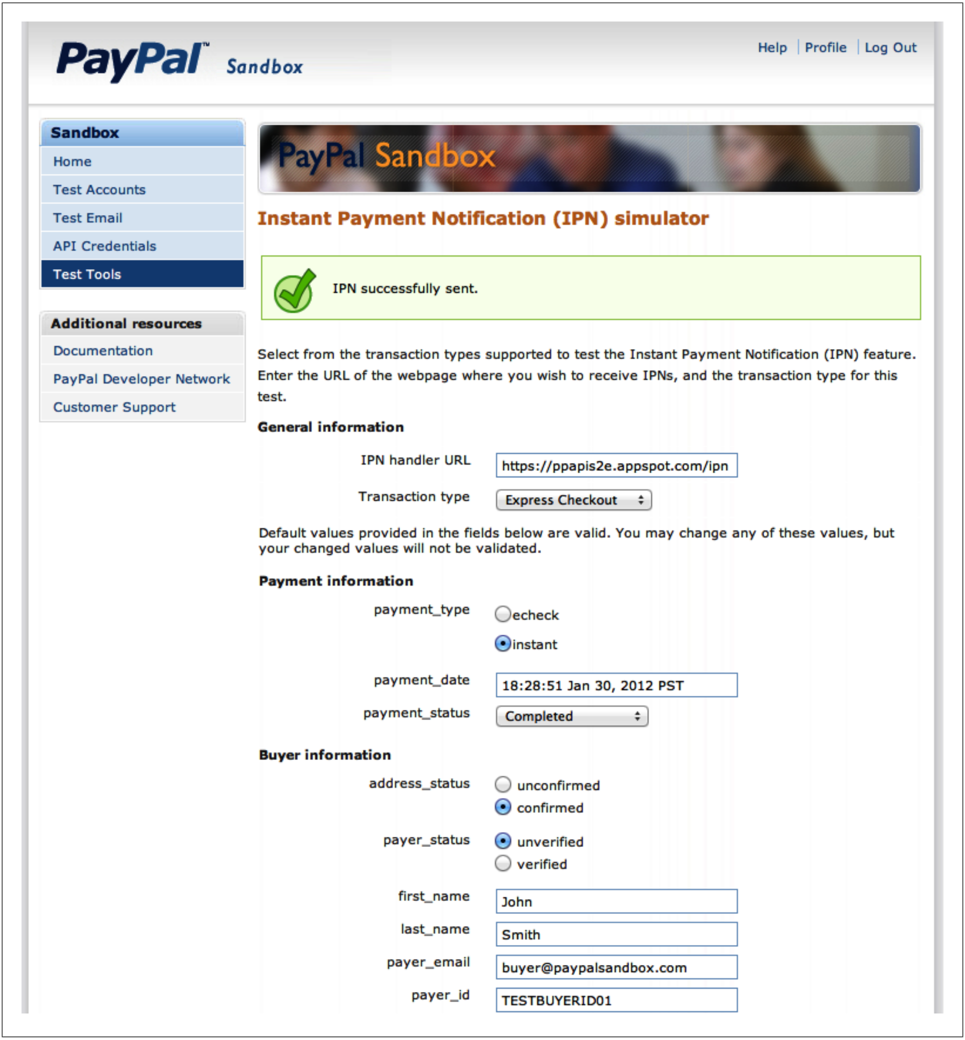
Whether it's on GAE or anywhere else, running a mail server during development can often be tricky and frustrating. You can run a local SMTP server that logs out messages it receives to the console using the Python `smtpd` module by executing the following command in a terminal:

```
$ python -m smtpd -n -c DebuggingServer localhost:1025
```

and configuring your local GAE development environment to send mail to it by starting the GAE Python Development Server from the command line with a few options, as follows:

```
$ /path/to/your/dev_appserver.py --smtp_host=localhost
--smtp_port=1025 /path/to/your/project
```

With some sample code in place, let's use the simulator that's available from within the PayPal Sandbox under the Test Tools tab to try out our IPN listener, as shown in Figure 6-3. After sending a notification with the simulator, you should see that it acknowledges that an IPN was successfully sent if it is able to successfully reach your handler and no internal server errors occur in your handler. Shortly thereafter, you should also receive an email with the IPN details since that's what the handler does. (If for some reason you do not receive the email, be sure to check your spam folder before digging into your application's logs, which are available through the GAE Dashboard.)



Instant Payment Notification (IPN) simulator

IPN successfully sent.

Select from the transaction types supported to test the Instant Payment Notification (IPN) feature. Enter the URL of the webpage where you wish to receive IPNs, and the transaction type for this test.

General information

IPN handler URL	https://ppapis2e.appspot.com/ipn
Transaction type	Express Checkout

Default values provided in the fields below are valid. You may change any of these values, but your changed values will not be validated.

Payment information

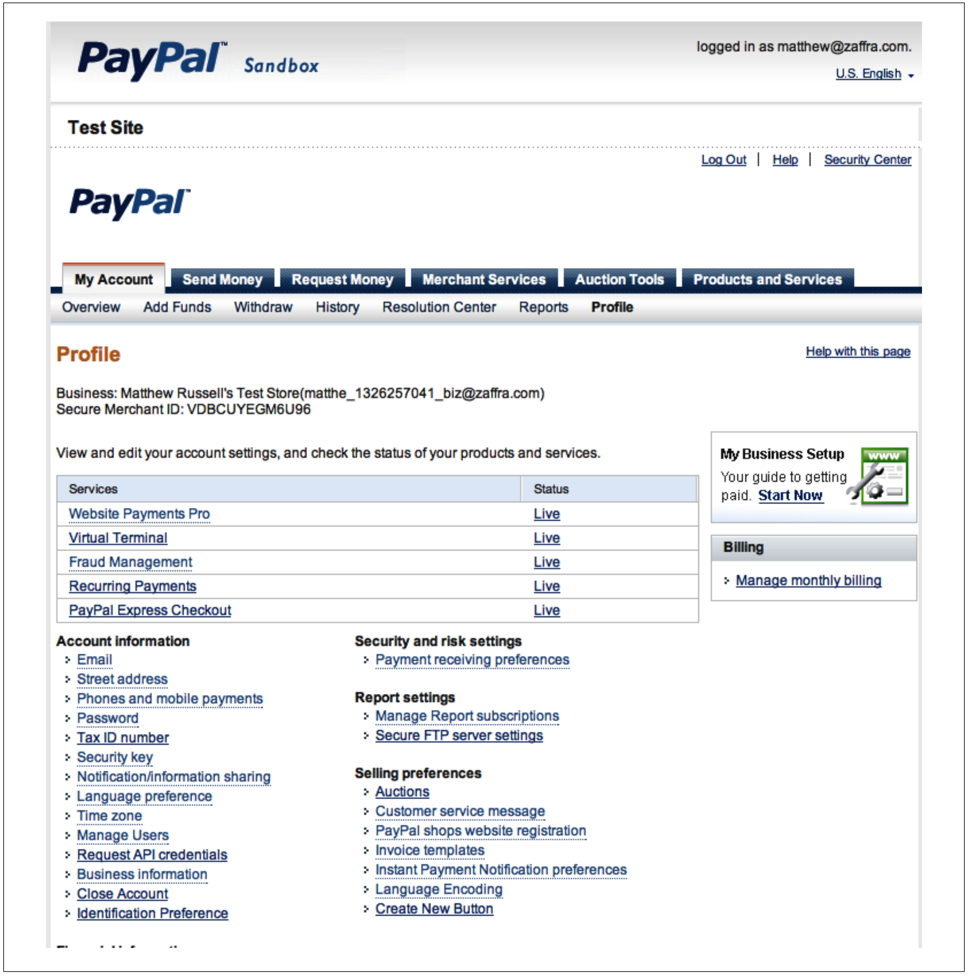
payment_type	echeck [✓ instant]
payment_date	18:28:51 Jan 30, 2012 PST
payment_status	Completed

Buyer information

address_status	unconfirmed [✓ confirmed]
payer_status	[✓ unverified] verified
first_name	John
last_name	Smith
payer_email	buyer@paypalsandbox.com
payer_id	TESTBUYERID01

Figure 6-3. Log in to your merchant account and select "Instant Payment Notification preferences" under the Profile tab in order to add a Notification URL

Now that you have some code in place that's capable of handling notifications from the IPN simulator, let's configure your merchant account for receiving IPNs so that you are able to automatically receive IPNs in response to payment transactions that affect that merchant account. Although it's possible to specify IPN notification URLs directly in API calls, perhaps the most common way to register for IPNs is to log in to a merchant account and specify a default URL under the merchant account's "Instant Payments Notification preferences" from within the Profile tab, as shown in Figures 6-4 and 6-5. Note that if you have multiple merchant accounts setup in your Sandbox environment, you'll need to ensure that you use the 3-token credentials associated with the



PayPal Sandbox

logged in as matthew@zaffra.com.

U.S. English

Test Site

Log Out | Help | Security Center

My Account | Send Money | Request Money | Merchant Services | Auction Tools | Products and Services

Overview | Add Funds | Withdraw | History | Resolution Center | Reports | Profile

Profile

Help with this page

Business: Matthew Russell's Test Store(matthe_1326257041_biz@zaffra.com)
Secure Merchant ID: VDBCUYEGM6U96

View and edit your account settings, and check the status of your products and services.

Services	Status
Website Payments Pro	Live
Virtual Terminal	Live
Fraud Management	Live
Recurring Payments	Live
PayPal Express Checkout	Live

My Business Setup
Your guide to getting paid. Start Now

Billing
- Manage monthly billing

Account information
- Email
- Street address
- Phones and mobile payments
- Password
- Tax ID number
- Security key
- Notification/information sharing
- Language preference
- Time zone
- Manage Users
- Request API credentials
- Business information
- Close Account
- Identification Preference

Security and risk settings
- Payment receiving preferences

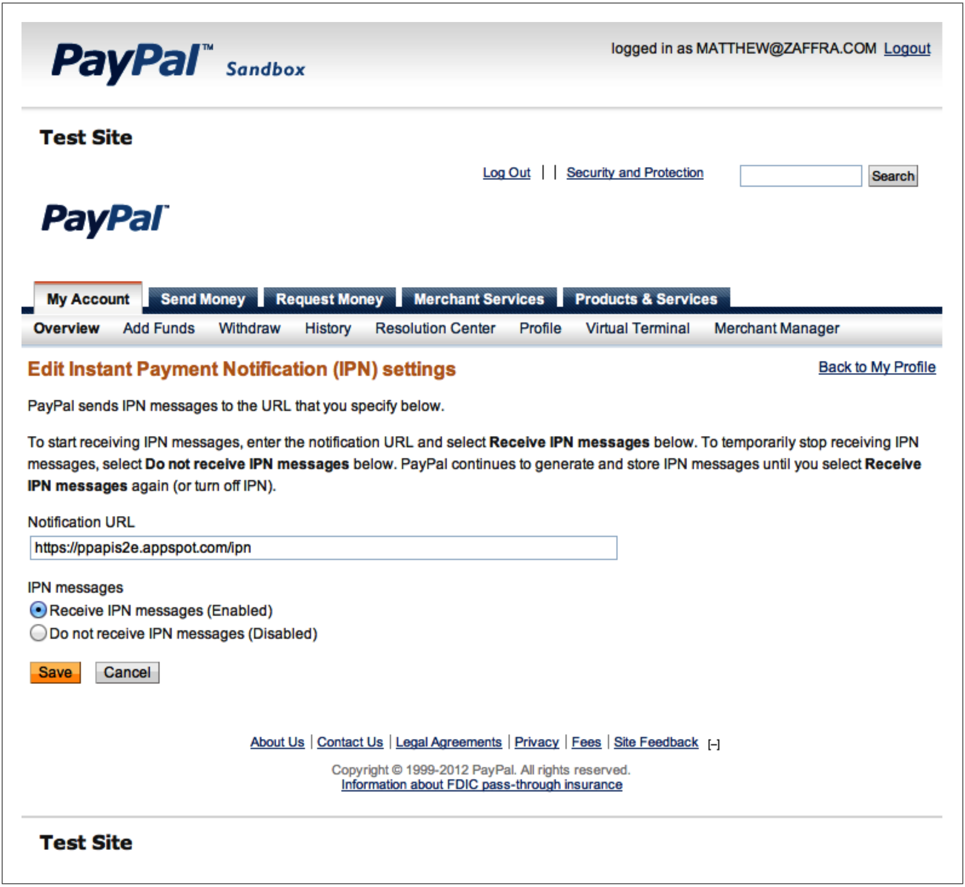
Report settings
- Manage Report subscriptions
- Secure FTP server settings

Selling preferences
- Auctions
- Customer service message
- PayPal shops website registration
- Invoice templates
- Instant Payment Notification preferences
- Language Encoding
- Create New Button

Figure 6-4. Log in to your merchant account and select “Instant Payment Notification preferences” under the Profile tab in order to add a Notification URL

same merchant account that you used to add the notification URL, and once you set up this IPN notification URL, PayPal will begin sending IPNs to it by default for all IPN-eligible transactions associated with those 3-token credentials.

Once you’ve deployed your IPN handler and registered a merchant account for receiving IPNs, you should start automatically receiving them whenever a successful transaction occurs that affects that merchant account. A recommended exercise for this chapter involves taking sample project code from a previous chapter and using it to trigger an IPN. Just be advised that once you register for IPNs, you will receive them for all IPN-eligible transactions affecting that account, and PayPal will continue resending them until you acknowledge or disable the IPNs from your merchant account profile.



Edit Instant Payment Notification (IPN) settings

PayPal sends IPN messages to the URL that you specify below.

To start receiving IPN messages, enter the notification URL and select **Receive IPN messages** below. To temporarily stop receiving IPN messages, select **Do not receive IPN messages** below. PayPal continues to generate and store IPN messages until you select **Receive IPN messages** again (or turn off IPN).

Notification URL: https://ppapis2e.appspot.com/ipn

IPN messages
[✓ Receive IPN messages (Enabled)]
Do not receive IPN messages (Disabled)

Save
Cancel
About Us | Contact Us | Legal Agreements | Privacy | Fees | Site Feedback [-]
Copyright © 1999-2012 PayPal. All rights reserved.
Information about FDIC pass-through insurance
Test Site

Figure 6-5. PayPal provides the ability to send all IPNs for a merchant account to a notification URL that you specify in your merchant account settings

Recommended Exercises

- Deploy the IPN listener code to AppSpot and run the sample code from Chapter 2 locally on your computer with the Google App Engine Launcher in order to log in and purchase a Tweet Relevance subscription using Express Checkout. After successfully performing the Express Checkout, your IPN listener should automatically receive an IPN that sends you an email.
- Transplant the `IPNHandler` featured in the sample code from this chapter into the sample code from another chapter, such as Chapter 2, in order to have a single, unified project file that features IPNs. Deploy this project to AppSpot and complete a payment transaction to trigger an IPN and ensure that it works as expected.

- Update the Tweet Relevance's `PaymentHandler` class from Chapter 2 to properly handle eChecks for an Express Checkout situation in which someone pays for a subscription with funds that must be drawn from a bank account because their PayPal balance is too low and they don't have a credit card on file.
- Modify the `PaymentHandler` class from Chapter 2 or Chapter 4 so that account transactions that are funded by eChecks are no longer rejected. (Recall that eChecks are not an option for digital goods purchases.) One promising scenario might be to optimistically assume that pending eCheck payments will clear successfully and initially credit accounts, but use IPNs to deactivate accounts and send users an email notification if the eCheck payment does not ultimately complete successfully.

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!

APPENDIX A

Overview of Tweet Relevance

PayPal APIs are of no use to an application that doesn't have revenue-generating potential, just as a technical book that's filled with abstract examples and void of useful sample code is of little use to the reader. As such, it seems appropriate that a technical book on integrating PayPal APIs should be packed with examples and sample code that illustrate how to integrate PayPal products into a somewhat realistic application. Teaching a reader how to get "up and running" is a somewhat delicate balance: it requires getting into messy implementation details with a specific toolchain and a realistic reference application while avoiding unnecessary complexity that hinders learning.

Tweet Relevance, the sample application that's referenced and built upon throughout this book, attempts to strike this balance. It's implemented in Python (one of the easiest-to-read programming languages), runs on Google App Engine (a web application platform that is mature and extremely well documented), and munges data from Twitter (an accessible and extremely rich source of information). Given that the scope of the book is about getting up and running with PayPal APIs, each chapter must maximally focus on PayPal products; however, in the interest of providing you with some sample code that's as useful and realistic as possible, Tweet Relevance provides a foundation that each chapter builds upon as a reference project.

The presumed problem that Tweet Relevance solves is information overload. The presumption is that even if a Twitter user very carefully curates and organizes a list of friends on Twitter, it can still be quite overwhelming to filter out the noise and keep up with the most relevant tweets. In other words, following people on Twitter does not scale very well. This is partially because keeping up with even a few dozen friends on Twitter can be a daunting task (especially when everyone gets chattier than usual), but also because tweets that appear in a user's home timeline are ranked chronologically, with no ranking heuristics applied to sort tweets by relevance. For example, although you might follow a person on Twitter because you really respect this individual's point of view on technology, you might not care at all what she intermittently has to say about politics, religion, or the environment. The value of a machine curating your tweets only becomes more apparent as you follow more and more people.

The choice of whether you treat the application logic for Tweet Relevance as a black box that you never really have to open or a new hobby that you invest a nontrivial amount of time improving and expanding is entirely up to you. The implementation is essentially stateless; is as lean and free from common third-party dependencies such as Django or other web frameworks as possible; and apart from minimally adapting a rich Ajax interface called TweetView[1] that supports touch gestures that Tweet Relevance itself treats as a third-party dependency, Tweet Relevance provides a no-frills user interface.

Before we start getting into the details, it may be helpful to take in the big picture by briefly reviewing the overall user experience for Tweet Relevance:

- User accesses Tweet Relevance
- User clicks on a "login with Twitter" button
- Tweet Relevance redirects the user to Twitter for authentication and authorization via OAuth
- User authorizes Tweet Relevance to access tweets and basic account information (which also serves as an authentication mechanism)
- Twitter redirects back to Tweet Relevance
- Using information available from Twitter (via OAuth), Tweet Relevance establishes a minimal account for the user, accesses the user's tweets, ranks them by relevance, and stashes them in a session object
- Tweet Relevance serves up the ranked tweets in TweetView

If you haven't already, the best thing for you to do right now is to get minimally familiar with GAE, as introduced in Chapter 1. Check out the sample code and try running it. The *README* file talks you through the steps involved, which includes creating a Twitter application that can allow Tweet Relevance to access your Twitter account through OAuth.

Understanding Tweet Relevance's AppHandler

In GAE parlance, a RequestHandler—often called a *handler*—is a class that services URL requests as mapped by the WSGIApplication in *main.py*. To illustrate, let's consider the *main.py* for Tweet Relevance, as shown in Example A-1.

Example A-1. Tweet Relevance - main.py

```
# Minimal GAE imports to run the app
```

1. Many thanks to SitePen for releasing high-quality, useful, instructive, and liberally licensed sample code such as TweetView.

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp import util

# Application specific logic

from handlers.AppHandler import AppHandler

# Logic for interacting with Twitter's API and serving up data, etc.

def main():

  application = webapp.WSGIApplication([

                                        # AppHandler URLs

                                        ('/(app)', AppHandler),
                                        ('/(data)', AppHandler),
                                        ('/(login)', AppHandler),
                                        ('/', AppHandler)
                                       ],

                                       debug=True)
  util.run_wsgi_app(application)

if __name__ == '__main__':
  main()
```

Tweet Relevance delegates responsibility for all of its four possible URL requests to a class called `AppHandler`. We won't dig into all of the nitty-gritty implementation details of `AppHandler`, but it is helpful to have a basic understanding of how each URL in this GAE application is serviced. The skeleton for the `AppHandler` class looks something like this:

Example A-2. Skeleton for AppHandler

```
class AppHandler(webapp.RequestHandler):

  def get(self, mode=""):

    if mode == "app":

      # ...

    elif mode == "data":

      # ...

    elif mode == "login":

      # ...

    else: # root context

      # ...
```

Hence, the `WSGIApplication` parses the tuples in its list argument and passes on values to `AppHandler`'s `get` method as a named argument called `mode` for GET requests. Whatever else happens in the application logic is entirely up to your imagination and hard work. Let's now take a closer look at Tweet Relevance's public APIs that enable this user experience by examining each of these URLs in greater detail:

/
: A GET request to the root context of the application displays a "login with Twitter" button, and when the user clicks on the button, it triggers a /login request. Twitter exposes an authentication API by way of OAuth, an industry standard for allowing web applications to authenticate users and take actions on their behalf without requiring them to give up precious username and password combinations. For Tweet Relevance, authenticating with Twitter via OAuth makes a lot of sense because in addition to OAuth being the only way that we can fetch all the data we'll need to implement the application logic, leveraging Twitter's OAuth prevents the application from needing to handle the mundane details associated with account management and essentially provides these services as a freebie. Figures A-1 and A-2 and illustrate the login flow.

> OAuth is an incredibly interesting yet tangential topic to the fundamental aims of this book. There is ample reading material about it on the Web, so a discussion will not be regurgitated here. For a very thorough treatment of the topic, see also *Programming Social Applications* by Jonathan LeBlanc (O'Reilly, 2011).

/login
: A GET request to the /login context immediately redirects to Twitter for authentication via OAuth. After the user logs in and authorizes the application, Twitter redirects back to /app and includes some important OAuth query string parameters in the query string that the application is responsible for parsing out so that it can request data from Twitter's API about the authenticating user.

/app
: Twitter redirects the user back to /app by means of a GET request once the user authorizes Tweet Relevance. The vast majority of the application's logic is handled in /app to include creating a user's account upon initial login, decrementing the number of login requests that are remaining each time the application is accessed, and accessing data in the user's Twitter account in order to implement a heuristic that ranks tweets contained in the home timeline by relevance. Once /app has ranked a user's tweets by relevance, it uses GAE's memcache to store them away temporarily in what's essentially a minimal session implementation and immediately redirects the client to an Ajax-enabled user interface that fetches the ranked tweets by invoking /data along with the session identifier.

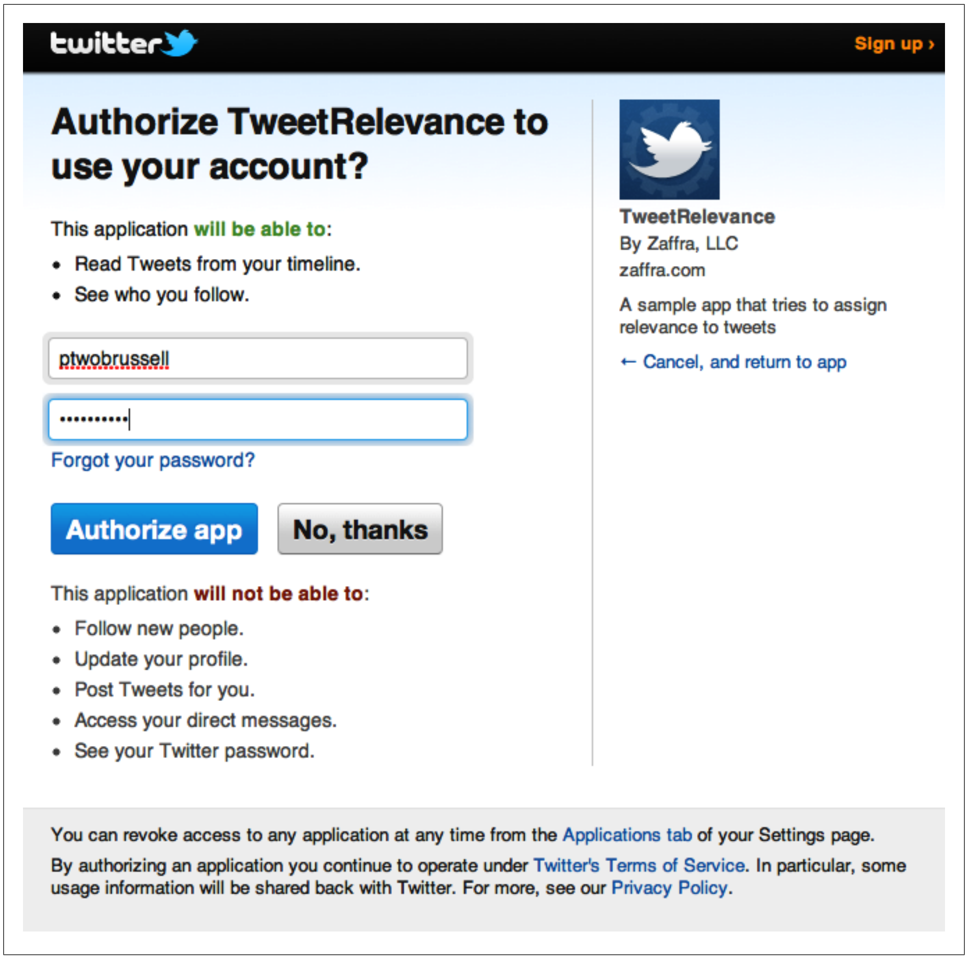
/data
: The Ajax client code issues a GET request on /data and passes in the session identifier to fetch ranked tweets so that it can display them to the user. Figure A-3 displays the user interface for Tweet Relevance, which powers its display data from /data.

Figure A-1. The root context of Tweet Relevance invites users to log in via Twitter.

One lingering question you may have at this point is how the heuristic is computed that ranks a user's tweets by relevance. The default implementation is actually nothing more than a trivial starting point: it simply computes a frequency distribution of terms appearing in tweets that have been marked as favorites and uses these terms to rank tweets appearing in the home timeline. Clearly, the logic involved in ranking tweets by relevance would be the "secret sauce" of *your* application should you choose to extend this sample project code to truly be worthy of revenue generation. This logic really is the core value proposition of the application, and there's quite literally no limit to the number of interesting things that you could try here. The default implementation is simply a placeholder for your own creative ideas. It will not make you a million dollars —but if you come up with a compelling way to rank tweets and create a backend that scales well, you might just be able to earn yourself a million dollars!

Recommended Exercises

- Complete the (official) Python tutorial.
- Review and execute the examples in the Getting Started with Python documentation for GAE.
- Check out the TweetView tutorials if you're interested in how to create rich Ajax clients.
- Execute the sample code for the application (this involves establishing a Twitter account with followers).
- Polish the application by defining some stylesheets for the templates.
- Expand upon and streamline the login flow by using session cookies.



Authorize TweetRelevance to use your account?

This application will be able to:

- Read Tweets from your timeline.
- See who you follow.

ptwobrussell

••••••••••

This application will not be able to:

- Follow new people.
- Update your profile.
- Post Tweets for you.
- Access your direct messages.
- See your Twitter password.

TweetRelevance
By Zaffra, LLC
zaffra.com

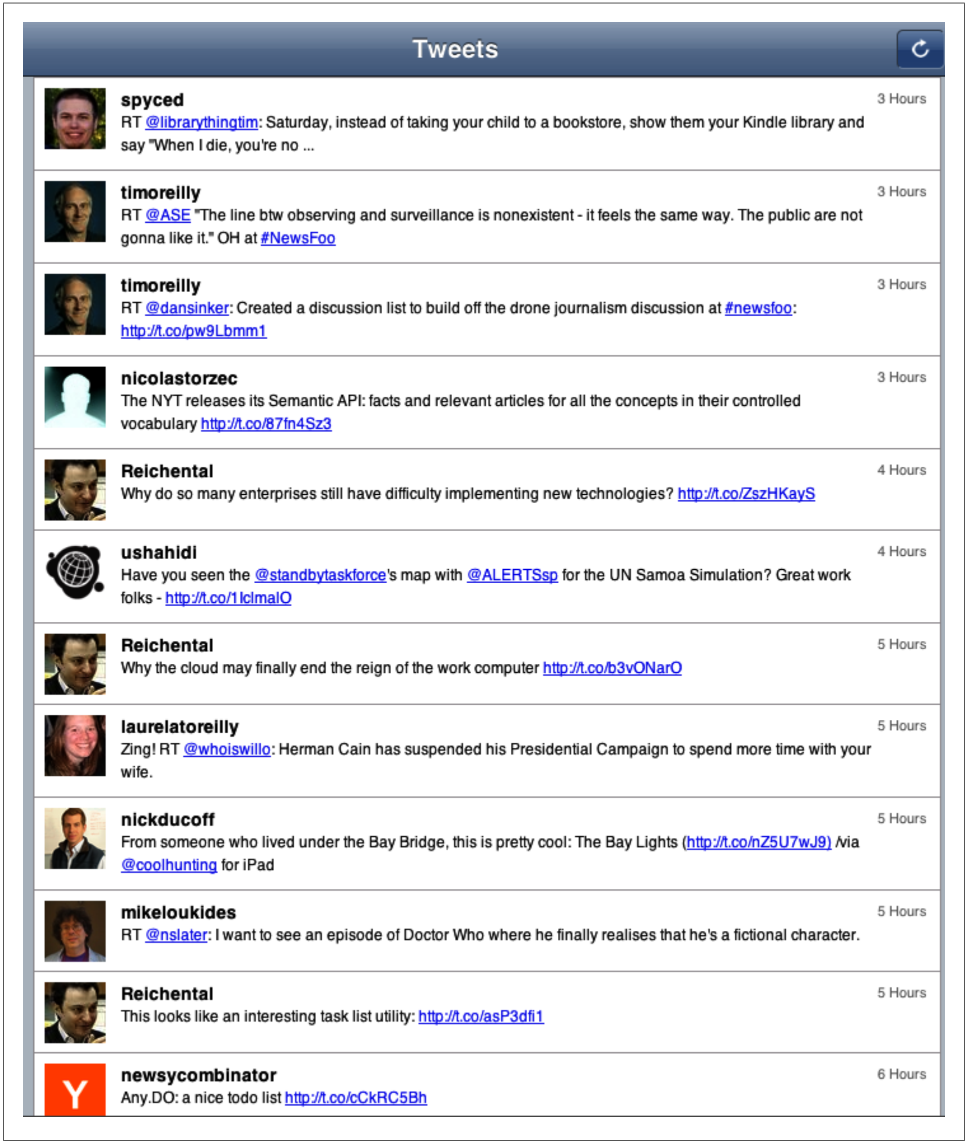
A sample app that tries to assign relevance to tweets

You can revoke access to any application at any time from the Applications tab of your Settings page.

By authorizing an application you continue to operate under Twitter's Terms of Service. In particular, some usage information will be shared back with Twitter. For more, see our Privacy Policy.

Figure A-2. Clicking the Login button triggers /login, which redirects to Twitter for authentication and authorization. Twitter redirects back to /app and passes along query string parameters that Tweet Relevance can use to access the user's information such as email address and tweet data

All sample code for this book is available online at GitHub and is conveniently organized by chapter. Although much of the code is included in the text of this book so that it's as instructional as possible, always reference the latest bug-fixed code on GitHub as the definitive reference for sample code. Bug tickets and patches are welcome!



Tweets

spyced 3 Hours
RT @librarythingtim: Saturday, instead of taking your child to a bookstore, show them your Kindle library and say "When I die, you're no ...

timoreilly 3 Hours
RT @ASE "The line btw observing and surveillance is nonexistent - it feels the same way. The public are not gonna like it." OH at #NewsFoo

timoreilly 3 Hours
RT @dansinker: Created a discussion list to build off the drone journalism discussion at #newsfoo: http://t.co/pw9Lbmm1

nicolastorzec 3 Hours
The NYT releases its Semantic API: facts and relevant articles for all the concepts in their controlled vocabulary http://t.co/87fn4Sz3

Reichental 4 Hours
Why do so many enterprises still have difficulty implementing new technologies? http://t.co/ZszHKayS

ushahidi 4 Hours
Have you seen the @standbytaskforce's map with @ALERTSsp for the UN Samoa Simulation? Great work folks - http://t.co/1IclmalO

Reichental 5 Hours
Why the cloud may finally end the reign of the work computer http://t.co/b3vONarO

laurelatoreilly 5 Hours
Zing! RT @whoiswillo: Herman Cain has suspended his Presidential Campaign to spend more time with your wife.

nickducoff 5 Hours
From someone who lived under the Bay Bridge, this is pretty cool: The Bay Lights (http://t.co/nZ5U7wJ9) /via @coolhunting for iPad

mikeloukides 5 Hours
RT @nslater: I want to see an episode of Doctor Who where he finally realises that he's a fictional character.

Reichental 5 Hours
This looks like an interesting task list utility: http://t.co/asP3dfi1

newsycombinator 6 Hours
Any.DO: a nice todo list http://t.co/cCkRC5Bh

Figure A-3. Tweet Relevance displays tweets from a user's home timeline in typical fashion.

APPENDIX B

Mobile Payment Libraries (MPLs)

Overview

If you've read this book cover to cover, you now know that it was designed to get you up and running with a broad array of popular PayPal products and a common technology set based upon GAE and Python. Although PayPal's Mobile Payment Libraries (MPLs)—which allow you to create in-app purchases for iOS, Android, and BlackBerry—don't fit the given scope and focus of the book because they require specialized development environments such as Eclipse and XCode as well as additional programming languages such as Java and Objective-C, you should definitely know that MPLs exist and what they can do to help you be successful in your commerce strategy. This brief appendix merely attempts to provide a shallow orientation and point you to some valuable resources online that can help you get up and running with MPLs. Entire books could be written on learning iOS and Objective-C, for example, with an underlying theme of using the corresponding MPL to implement an application involving mobile commerce.

In short, an application that employs MPLs allows you to embed a "Pay with PayPal" button *natively* within the iOS, Android, or BlackBerry application you're developing and provides you with an easy-to-use software development kit (SDK) that provides views for logging users into their PayPal accounts and processing payments. With regard to implementation details, there truly is minimal hassle involved in integrating MPLs into an existing app, and in many circumstances, there can be less work involved in integrating an MPL into a native mobile application than in integrating a product like Express Checkout into an existing web application.

Should I Use MPLs or Mobile Express Checkout (MEC)?

Recalling from "Mobile Express Checkout (MEC)" on page 44 that an Express Checkout flow "just works" on most modern mobile devices capable of browsing the Web, such as Android and iOS devices, you now have additional options that allow you provide in-app purchases without compromising the integrity of the user experience, which can really hurt your conversion rates. As a rule of thumb, you shouldn't look at MPLs versus a web-based Mobile Express Checkout (MEC) (a freebie that comes along with a standard Express Checkout implementation) as an either-or type of decision. The two solutions are orthogonal to one another and are usually very complementary in the way that they allow you to broaden your potential reach. For example, if you have an existing web application or plan to create a website or application that accepts payments, then you should target users regardless of whether they're using a laptop, tablet, or mobile phone. Thus, Express Checkout may be a great option since it trivially provides a seamless mobile experience that involves no additional work on your behalf. However, if you're building or already have a native app, then you should also plan to process payments in the app using MPLs. If your application lends itself to both the Web and to specific mobile platforms, then you'll inevitably end up using both options. Figure B-1 illustrates a sample application that uses MPLs to trigger a payment flow to PayPal.

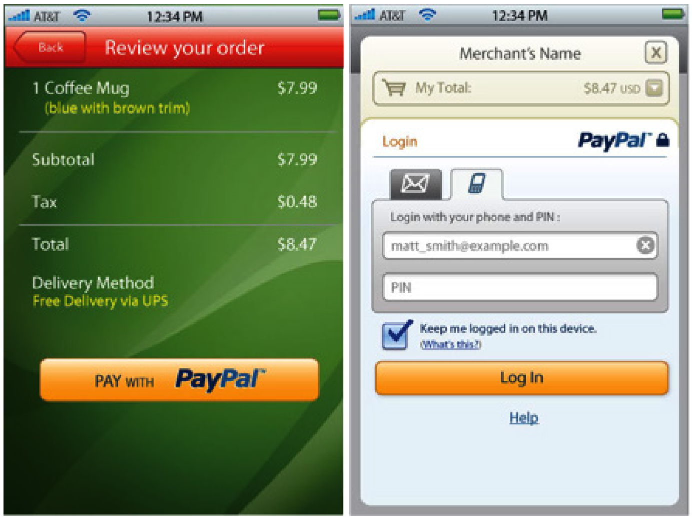


Figure B-1. MPLs provide a way to natively embed PayPal payments into your iOS, Android, and BlackBerry applications

In addition to the standard, more general-purpose MPLs that have been discussed in this Appendix, PayPal also offers an additional MPL called the Mobile Express Checkout Library (MECL), which can streamline the implementation of an Express Checkout flow from within a mobile application's web view. MECL is essentially the way to kick off an Express Checkout from within a mobile application and return control to your application when it completes. MECL also provides a simple way to launch a Mobile Express Checkout from a website. Although somewhat confusing, the MECL is distinct from a Mobile Express Checkout (MEC) in that the MECL is a library that kicks off an

MEC. Even though there's a lot of overlap in the verbiage, it's a bit of an apples and oranges comparison in that one is a library (that kicks off a payment flow) and one is a product that provides the payment flow itself.

Recommended Exercises

- Bookmark PayPal's canonical starting point for Mobile Payments Libraries, which contains links to detailed and up-to-date documentation on the SDKs for supported mobile platforms.
- If you're interested in developing an iOS application using MPLs, take a look at the sample code for Inquire, a very well-documented mobile, social, and local application that uses MPLs to process payments. [With regard to MPL integration, you may be particularly interested in the PayPalViewController class.]
- If you're interested in developing an Android application using MPLs, take a look at the sample code for YardSale, an application that scans QR Codes and uses MPLs to process payments. [With regard to MPL integration, you may be particularly interested in the ItemDetails class.]
- If you're truly ambitious, port Tweet Relevance to iOS, Android, or BlackBerry and use MPLs to handle making payments.

About the Author

Matthew A. Russell, Vice President of Engineering at Digital Reasoning Systems and Principal at Zaffra, is a computer scientist who is passionate about data mining, open source, and web application technologies. He's also the author of *Dojo: The Definitive Guide* (O'Reilly).