Zippy Documentation

Release 0.1

Mozilla

February 24, 2016

Contents

Please note: this project is currently unmaintained and is not (or soon will not) be in active use by Mozilla.

This is the Mozilla reference implementation of a payment processor that would hook into a payment provider for navigator.mozPay() such as WebPay. It is a reference implementation of Marketplace payment providers specification.

It shows the API, endpoints, and authorization formats a payment processor should implement in order to integrate with the Firefox Marketplace via WebPay (and Solitude behind the scenes). Hooking into WebPay as a payment processor is much simpler than implementing a full-on payment provider to spec.

These docs are also available as a PDF: https://media.readthedocs.org/pdf/zippypayments/latest/zippypayments.pdf

Zippy provides you with a working example. It's important to note that underneath Zippy does not actually do anything other than record some transaction information to allow APIs to work. Specifically it does not:

- try and actually charge carrier billing
- try and charge a credit card
- process any money at all

That is all faked out. It's up to the individual payment processor to implement that.

CHAPTER 1

Do I have to use node?

No. You should not take our reference implementation of Zippy and use it. This is not production code.

The point of this is not to provide any code that you should run. Instead it is to **demonstrate and document a common API for payment processors**.

1.1 So how should I use Zippy?

Play with it, look at the API. Use it as a sample. Then use whatever software tools, frameworks and methodologies you use in development and build your own version of Zippy.

Your implementation should implement the Marketplace payment providers specification.

If you build something that has:

- the same API end points
- uses OAuth for authentication
- takes the same inputs in the same formats (e.g.: REST over HTTP)
- responds with the same outputs (e.g.: REST over HTTP)
- copes with errors in the same way

Then you will have a **zippy compatible** implmentation and all Mozilla has to do is change the configuration of the Firefox Marketplace and we'll plug into your implementation of the API.

1.2 How do I use the templates?

The templates are written in to work with the Mozilla Zippy implementation. That likely won't work for other people, you likely have a different HTML templating library. We haven't found a good solution for this yet. But Zippy gives you the HTML, the CSS, the JS and the localisations. That's a pretty good start.

1.3 How do I make this better?

This is an open source project, we welcome any and all feedback. We certainly haven't implemented all the APIs that could possibly exist. We don't know all the answers. We'd love to work with anyone using Zippy and make this better.

Mailing list: https://lists.mozilla.org/listinfo/dev-marketplace

It is licensed under the Mozilla Public License v2.0 and contributions are more than welcome.

Source: https://github.com/mozilla/zippy/

Bugs: https://bugzilla.mozilla.org/show_bug.cgi?id=905736

CHAPTER 2

Contents

2.1 Installation

This section is for developers who wish to make enhancements to Zippy or just to try the code out in a development environment.

2.1.1 Installing Zippy

You need NodeJS 0.10.5 or greater. Run this to install the deps:

```
npm install
```

For convenience, you may want to give all local node scripts priority on your path, like this:

```
export PATH="./node_modules/.bin/:${PATH}"
```

During development, you'll also want to install grunt-cli globally:

```
npm install -g grunt-cli
```

This will put the grunt command in your system path, allowing it to be run from any directory.

Note: Installing grunt-cli does not install the Grunt task runner! The job of the Grunt CLI is simple: run the version of Grunt which has been installed next to a Gruntfile. This allows multiple versions of Grunt to be installed on the same machine simultaneously.

Create a local config file and fill in some values like `signatureKeys`:

```
cp lib/config/local-dist.js lib/config/local.js
```

To start a development server type this:

```
grunt start
```

You can then browse the site at http://0.0.0.0:8080 (use the `--noauth` option in case you don't want to pass OAuth headers at each request).

If you want to change the port run:

```
grunt start --port=9999
```

grunt start runs both the local server and watchs for changes. At the moment this auto runs:

- stylus
- jshint

See *Testing* for instructions on how to run tests.

2.1.2 Building docs

Unlike the site itself, the documentation system uses Python and Sphinx. To build the documentation from source create a virtualenv then install the requirements with pip:

```
pip install -r docs/requirements.txt
```

Build the docs like this:

```
grunt docs
```

Browse the docs from `docs/_build/html/index.html`.

2.1.3 Updating Bower resources

Bower is a package manager for the web. All it does is pull in versioned client deps into a *bower_components* dir.

It's very similar to npm. So a bower.json should seem very familiar to you if you've used a package.json for npm.

Bower manages our third party libs for the client. If you want to update those libs first update bower.json with the new libs you want to use.

Next if you've added a new client-side dep. You need to update some config. Because most bower package authors don't yet use the ignore feature, we're using grunt-bower-task to copy the necessary files under *media/lib*. This saves us having to server a ton of tests and other cruft above and beyond the lib files.

If you want to customise how that works then see the *exportsOverride* in the bower.json. This points at the files by type (e.g JS or CSS) so that only the referenced files will end up in the lib dir.

To see the general configuration take a look at how grunt-bower-task is configured in Gruntfile.js.

If you need additional guidance the grunt-bower-task docs should have what you need.

Once the configuration is complete running *grunt bower:install* should copy the new lib files into *media/lib*.

Next. You can update the requirejs config in *media/js/main.js* if you're using JS in order to be able to reference the files in other scripts.

2.1.4 Sample server

A sample zippy server is running at https://zippy.paas.allizom.org, that you are free to use. There are no guarantees on uptime, this is not a production server.

2.2 Using

Zippy has a JSON-based HTTP API to programmatically interact with the payment system.

To ensure you retrieve results in JSON format, make sure to add an accept header of `application/json` to all API requests. For example:

```
curl -H "Accept: application/json" -X POST ...
```

2.2.1 Authentication

Zippy requires authentication on all requests. To make development and testing with the zippy reference implementation, it can be run without authentication, but this should never occur in production. To run without authentication:

```
grunt server --noauth
```

The authentication uses OAuth 1.0a (IETF specification) with the 0-legged scenario, the client of the API has to know 2 settings prior to performing a request:

- the `consumer key`
- the `consumer secret`

Each of these parameters are 16 chars long strings delivered by the implementor of Zippy. You have to pass those parameters and the regular OAuth signature of parameters within the `Authorization` header to be able to perform a request. Here is an example of the full header (splited across multiple lines for lisibility but must be kept as a one-liner header):

```
OAuth realm="Zippy",oauth_signature="wpMJyt18lPuVIzymgL4WHHVao7A=",
oauth_consumer_key="dpf43f3p2l4k3l03",oauth_nonce="notimplemented",
oauth_signature_method="HMAC-SHA1",oauth_timestamp="notimplemented",
oauth_token="notimplemented",oauth_version="1.0"
```

Given the 0-legged scenario, the user is not involved in this workflow, that's why you don't have to deal with the classic OAuth authentication token.

To set the config alter *lib/config/local.js* to read like the following, replacing the keys, secret and realm with appropriate values:

```
module.exports = function(conf) {
    // Add a non-empty value to the zero-ith key.
    conf.signatureKeys = {0: 'A-SECRET-KEY'};
    conf.OAuthCredentials = {
        consumerKey: 'A-CONSUMER-KEY',
        consumerSecret: 'A-CONSUMER-SECRET',
    };
    conf.OAuthRealm = 'SERVER-DOMAIN';
};
```

The consumer key and secret values will need to be used by the client to generate the approriate OAuth tokens and should be stored securely on the client.

2.2.2 Resources

Each resource always returns 3 additional parameters:

- a `resource_uri`: a relative URI to access the resource,
- a `resource_name`: the plural name of the resource,
- a `resource_pk`: the unique identifier of the resource.

2.2.3 Errors

TODO: How errors are displayed and handled.

2.3 l10n and i18n

Zippy uses i18n-abide for l10n purposes. It provides gettext functions for the application and also provides the ability to use a debug locale for testing l10n.

To create, extract, merge and compile locales we are using grunt-i18n-abide which is a set of grunt tasks to run the localisation tools.

To use grunt-i18n-abide you'll need to make sure you have your distro's standard tools installed. You should install Gnu gettext to get *msginit*, *xgettext*, and other tools.

2.3.1 Creating new locales

First add the language you want to support to the *supportedLanguages* list in 'lib/config/default.js'

Once this is there run *grunt abideCreate* and the locale dir will be automatically created along with a *.po* file.

2.3.2 Extracting new strings

To extract new strings run *grunt abideExtract*. This should update the template.

2.3.3 Merging new strings into existing locales

To merge newly extracted strings into existing locales run *grunt abideMerge*.

2.3.4 Compile locales

Currently Zippy only needs json files to be compiled for l10n purposes. To that end simply run *grunt abideCompile*.

2.3.5 General notes

Always check what's created is what's expected. Version control can really help with this. If you have any problems and what's created isn't what you expect please file a bug with the relevant project.

2.4 Payment

When processing the payment, Zippy simulates charging money for digital goods. When finished, it redirects to the payment provider (such as WebPay) and sends a post notification. Please see the specification for a diagram of the flow. Exactly what will happen here depends upon the payment processor and the configuration. After these steps have been completed, Zippy will return to the success or error URL. See the *How Redirects Work* for details.

A real payment processor would probably do things like this:

- Authentication: set up a user for billing, perhaps with SMS authentication.

- Direct billing: place a charge on a user's mobile bill.
- Credit card billing.

2.4.1 How Redirects Work

When Zippy completes a transaction it redirects to the original success or error URL (see transactions API for how those are defined). In the case of success, the application that began payment would respond with the HTML/CSS/JS needed to dispurse the goods.

A few query string parameters are added to the URL that you can use to reconcile the payment.

ext_transaction_id This is the original (external) transaction ID that was submitted to the transaction API as `ext_transaction_id`.

error For error redirects only, this string will indicate the type of error. Example: `CC_ERROR`. Note: it's up to each payment processor to define their own error codes.

Example: `https://site/payments/success/?ext_transaction_id=XYZ`

Note: A signed query string notice must be verified before any of the values can be trusted.

2.4.2 How Post Notifications Work

When Zippy completes a transaction it not only redirects to the success/error URL, it also sends a post notification in the background. There are some edge cases in web user agents that could interrupt a redirect request so post notifications are generally more reliable. An application processing a payment should expect to continue the payment flow after redirect but should use post notifications as an additional measure to reconcile payment results.

An application configures its callback URLs when beginning a transaction. Zippy will post a single parameter called `signed_notice` to either the callback success or error URL. This parameter contains a URL-encoded, signed notice that must be verified and then URL-decoded.

The notice query string has the same parameters as the one sent in a *redirect*.

Style guide

Zippy contains a full style guide containing the CSS, HTML and JS to be used on a page. It will also contain localisations.

If a page has been implemented in zippy, then it can be used by a payment provider by copying and pasting over the code into the existing payment providers framework. It might be worth payment providers thinking about this step as it creates a bit of a long term maintenance issue.

The style guide is accessible in your zippy checkout, or here:

http://zippy.paas.allizom.org/styleguide

2.4.3 Translations

All the pages are translated. For a status of the translations see:

https://localize.mozilla.org/projects/zippy

The translations are available in the zippy repository:

https://github.com/mozilla/zippy/tree/master/locale

2.5 Zippy Config

Config works by overlaying additional rules on top of the base set.

Keys are looked up when you ask for them using a getter and values are provided based on the following criteria:

1. If *NODE_ENV* is not 'test' use local config value if that key exists.
2. Use key in *NODE_ENV* config if it exists and has been setup in *config/index.js*.
3. Use key in defaults.js

Local overrides are possible unless the *NODE_ENV* environment is test

2.5.1 Adding a new config

To add a new config e.g: 'edge' create a new config file in lib/config called 'edge.js'

In it define the keys you want to override. Totally new keys are allowed but you'll get a warning if it doesn't exist in the defaults conf.

e.g:

```
module.exports = {
    showClientConsole: true,
};
```

Lastly update lib/config/index.js to add the require the new file adding it to the overall config object.

e.g:: config.edge = require('./edge');

Note: If you're adding something that shouldn't be committed to the repo then you'll want a try/catch around the require see how the local file is required for an example. (Now would be a good time to add it to .gitignore!).

2.5.2 Using the config

Using the config is just a case of setting the *NODE_ENV* environment var before starting up the app. E.g:

```
NODE_ENV=test nodejs main.js
```

2.6 Testing

2.6.1 Running Automated tests

To start the tests run:

```
grunt test
```

This will run both jshint and nodeunit.

To test just one suite:

```
grunt test --testsuite products
```

This will only launch tests from the file *suite/products.test.js*.

To test just one test from the suite:

```
grunt test --testsuite products --test postOk
```

This will also run all tests that match:

```
grunt test --test postOk
```

2.6.2 Automated UI tests via Casperjs

Zippy has a test suite that runs tests against the web UI in a headless browser. To run UI tests you need **'casperjs'_** 1.1 or greater. With **'homebrew'_** on Mac OS X you can install it like this:

```
brew install --devel casperjs
```

From the root of Webpay and within your Python virualenv, run the tests like this:

```
grunt uitest
```

To hack on tests, add a file like `uitest/suite/test.*.js`. All JS files in that directory are run automatically.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search