Pipfit: Password Ignorant Personal Financial Tracker
Akshay Nathan
Advisor: Ruzica Piskac

Introduction:

Personal Financial Manager (PFM) applications have been around almost as long as personal computers. Recently, these applications have migrated online, in the form of popular sites like Mint. Mint and similar apps aggregate users’ income and spending data, and provide a host of visualization and analysis options. Moreover, many of the sites have additional features which allow for budgeting and goal setting, managing account alerts, and analyzing long term spending trends.

All of these features rely significantly on the quality of the data. Most of these apps require users to enter bank login information, and then use advanced screen-scraping techniques to collect transaction data. Sometimes this process is outsourced, and services like Yodlee or Plaid provide SaaS solutions to the neverending problem of reliably parsing and securely storing user information. Versions that do not require bank credentials force the user to manually download the transaction data from the banks, and then upload it to the application to be processed.

It is obvious that from a purely convenience oriented standpoint, scraping techniques are best for the user. However, scraping is computationally expensive, often violates the ToS of these banks, and the institution themselves have no reason to not arbitrarily change their online platforms, breaking the scrapers. Delegating this duty to outside services as discussed above is also expensive. From the banks’ perspective, the constant barrage of traffic from scrapers causes problems on their own servers. Lastly, from the users’ standpoint, many people are simply
uncomfortable providing their bank details to outside services, especially because the credentials must be stored in plaintext for the scrapers to login. A security breach here, could be devastating -- the same details which enable users to view transaction history and statements can also be used to arbitrarily transfer funds.

The existing solution to these problems, manually downloading and uploading the data, is simply a tremendous pain, especially when each version of these files only differs by a few transactions.

In order to address these and other problems, I proposed a novel PFM service which would aspire to the following three goals:

1. Requires no bank account information (usernames, passwords, or account numbers).
2. Automatically updates transaction history.
3. Open-source and can be self-hosted.

To reach these objectives, this new service does not rely on screen scraping or any type of post-authentication information. Instead, we depend on the “Alerts” features that all bank and credit card companies provide. For example, in my own Chase account, I can request Alerts any time a transaction, overall balance, transfer, withdrawal, or deposit exceeds a certain amount. By setting this value to any arbitrarily low number (Chase allows $.01), we can receive alerts for basically all bank activity through email. Moreover, because these alerts are transmitted insecurely through email, the bank already sanitizes any sensitive information, leaving just the raw transaction details without any of the security risks.

The rest of this paper will describe this effort in detail, including design decisions and implementation specifics. Moreover, as an added challenge, I decided to do this project in
Clojure, a language completely new to me, and I will also describe my experiences with building a multi-component web application in a functional language.

**Data Schema**

The notion of a transaction lies at the very core of this app. A transaction (see `pipfit/parser/transaction.clj`) represents activity within an account. Transactions have a type -- they can be withdrawals, transfers, payments, etc. Moreover, for each of these types, transactions are bundled with different sets of metadata, for example, transfers happen between two parties whereas withdrawals consist of simply an amount and a time. In order to represent this data structure in a flexible way, and in accordance with Clojure’s distancing from OO conventions, we represent transactions as maps. Each type (an enum) of transaction has a vector of associated fields, which allows us the flexibility to add types or fields at will, all while using the benefits of simple list comprehension in functional languages to quickly validate these maps.

The higher level data patterns are straightforward. Transactions belong to categories, eg. food, drink, travel, etc. Transactions also belong to accounts, which can be of a type, checking, savings, or credit, and are associated with a bank. Accounts belong to users, and are identified simply by the last four digits of the account number. This was chosen in order to preserve privacy, and also for convenience; most email correspondence includes only these digits.

**Data Collection**

Every user is assigned a unique alphanumeric ID. Users can either set their bank preferences to directly send alerts to user_id@pipfit.com, or more easily, create filters on their own email accounts which will auto-forward all alerts to this address. On the opposite end, a
server manages these incoming emails, parses out the relevant information, finds the associated user, and enters the transactions into the database. In the next section, we will discuss the parsing infrastructure, but first we will look to the two data sources included with the app.

The first method is the traditional mail server. One can set up a linux server with AWS, install a mail daemon, smtp client, and database, and use the included interfaces (see `pipfit/server/emailparser.clj`) to watch the filesystem for changes (incoming mail) and parse out the transaction information. This method aligns with our goal of self-hosting. If one were to go this route, they would control all data endpoints and it would definitely be the most secure. However, hosting your own mail server comes with a variety of issues. For one, spam filtering is a difficult problem, and any public facing addresses are at risk. Secondly, mail server configuration is not easy, and may be a deterrent to users. Therefore, I also included a second solution, which involves far less setup and is generally easier to handle.

Mailgun ([http://www.mailgun.com](http://www.mailgun.com)) provides services for companies to send and track mail, for applications like newsletters, alerts, or product offerings. Interestingly, they also provide a mail receiving platform, as long as you have your own domain. With the correct DNS records, mail sent to your address at user_id@pipfit.com, is automatically filtered for spam, and sent via HTTP to a user specified server (see `pipfit/server/mailgunserver.clj`). This server receives messages from mailgun over HTTP post, parses the contents and writes to the database. Users who wish to host the app themselves simply need to create a free mailgun account, and specify their endpoint to an instance of this server.
Lastly, if users wish to add any history from before they started using the app, we provide the ability to import OFX (Open Financial Exchange) files, the common file format for transaction data, into Pipfit (see \texttt{pipfit/parser/ofxparser.clj}). This, however, is not required.

**Parsing**

One of my main goals with this project was to adhere closely to convention among the clojure community of extremely simple, modular applications. With parsing, I wanted to make it as easy as possible for contributors to create and plug in their own parsers for their own bank sources.

To achieve this, I have specified a Parser interface (see \texttt{pipfit/parser/accountparser.clj}). Parsers have three main functions: they must identify messages from their associated account types, parse these messages, and also parse OFX files from their related institutions, which sadly differ widely from bank to bank. Parsers have basic identifying information, which also includes a version number, which is updated at each iteration of the parser. This version number enables contributors to quickly iterate on parsers when and if a bank’s email alert format changes, and more importantly, for the application to easily recognize these changes without breaking or damaging transaction history.

To add a parser, a contributor simply satisfies the interface, writes corresponding tests, and adds the parser to a vector (see \texttt{parserslist.clj}). On an incoming message, the application uses a dispatch function to greedily designate a parser by iterating through the identify functions. This design allows parsers to be added without changing any other code.

We also provide helper functions (\texttt{parser/helpers.clj}), to further standardize and smooth out the parsing process. The included reference parser for a Chase Checking account can be
found in (parsers/acctparsers/chasechecking.clj), and helps demonstrate how the parser abstraction helps overcome the relatively simple but still significant differences between email alert types, even within the same bank.

Client-Side

On the client-side, I decided to stick with Clojure by using Clojurescript, a subset of the language without the JVM additions which compiles to Javascript. By this point in the project, I had really began to appreciate the convenience of immutable data, as it forced me to evaluate my design decisions thoroughly and helped in debugging. Thus, I decided to use the Om (http://www.github.com/omcljs) framework, which mandates immutable state and discourages direct manipulation of DOM html elements.

The Om pattern is based on React (http://www.github.com/react) with some added features. Applications are described as a global, binding state, as well as components, which interact with and render this state. These components could be HTML tables, or Javascript animations. The major feature of Om is that instead of directly changing properties on these components, you can only change global state. This abstraction removes the necessity for callbacks, and provides asynchronous performance right out of the gate. Components now listen on channels to published state changes, which improves parallelization and thread safety. Perhaps most importantly, without callbacks code is much more self documenting and easily followable.

This being said, however, the transition to the Om mentality from experience in traditional web frameworks has been difficult. To stay true with the goals of my proposal, I have built the entire front-end of the application client-side. Users can authenticate via json with an
api on the server, and all additional data is also passed via json and rendered solely in the
browser. For example, transaction filtering (see dashboard.cljs) simply pulls as much data as
possible from the actual datastore, and then uses a series of functional steps to filter and modify
this data completely in the browser. The advantage here of Clojure is that I was able to use my
server side functions and data models on the client side with ClojureScript!

This decision furthers the broader goal of modularity. Simply the collection and parsing
aspects of this project would be useful by themselves, if a user wanted to build their own
front-end or maybe mobile app around this. By using standard json apis, we facilitate
extensibility, again without having to touch any of the server code.

Results

In this section I will address the deliverables outlined in my proposal.

- A better name
  - “pipfit” is better than “An Open-Source Personal Finance Manager”
- Modular, testable, working application with 2 different bank sources.
  - As discussed above, I have architected this app to be as modular as possible. I
    have also included two account parsers as reference, for the Chase Checking and
    Chase Credit account types respectively.
- Aesthetically pleasing UI.
  - Unfortunately, I did not have as much time to work on style as possible. This,
    however, will soon be fixed as I plan to continue working on this project.
- Fully in Clojure / ClojureScript.
  - Everything in the app including configuration is written in Clojure.
- Supports OFX importing.
  - I have included an OFX parser for Chase files as well as general OFX parsing
    interfaces for future parsers.
- Support for different types of account activity.
  - Currently, pipfit supports Debit Transactions, Withdrawals, Transfers, Credit
    Card Payments, and Deposits.
- Supports multiple users.
As discussed above, I have created an authentication system where users can create accounts and manage individual data.

Unfortunately, I was not able to reach all my goals, primarily because it took me much longer to learn and get used to Clojure and the various new frameworks than I anticipated. Functional programming, especially in applications like Pipfit, which have to manipulate state by interacting with external data sources, really forced me to completely change the way I think about building applications, definitely for the better.

**Future**

This app has already been extremely useful for me and I will definitely continue working on it. For one, I want to improve the filtering and categorization features, which I did not completely get to build out. I still believe that my original plan of a regular expression based category system has the potential to be extremely useful.

I also want to integrate d3.js ([http://www.github.com/d3](http://www.github.com/d3)) to add visualizations and graphs to further shed light on spending trends. Lastly, as discussed above, I plan to improve on style and aesthetics.

**Acknowledgements**

I want to thank my advisor, Ruzica Piskac, for advising me and allowing me to switch projects relatively late in the semester.

I also want to acknowledge all the open source work that this project has benefited from, some of which is named above. A full list can be conveniently found in the project.clj file of the source tree, which includes all external dependencies.