

Bacon: Final Report

Fabian Terh Jun Wei | Travis Ching Jia Yea | Zhang Lizhi

Requirements

Overview

Bacon is a gamified expense tracking application to make personal finance fun, frictionless and easy.

There are many expense trackers on the market, but they are often clunky, unintuitive, difficult to use, provide a bad user experience, and often intimidate users with too many charts, numbers, and options. As a result, users may be put off from recording transactions accurately in a timely fashion, resulting in inaccurate transaction histories - which defeats the entire point of an expense tracking application. We want to create an expense tracker that is both powerful yet frictionless in user experience that is a breeze to use.

Bacon is built on the *keep it simple, stupid* design philosophy. We believe that *simple is beautiful*, and that powerful and flexible features should not be at the expense of (pun intended) user experience (ease of use).

At a glance, these are the key features of Bacon:

- Budget overview
- Transaction history overview
- Monthly net income trend analysis
- Expenditure breakdown by tags
- Expenditure breakdown by location
- Powerful and flexible transaction recording
- Smart transaction predictions
- Smart location-driven prompts

Revised Specifications

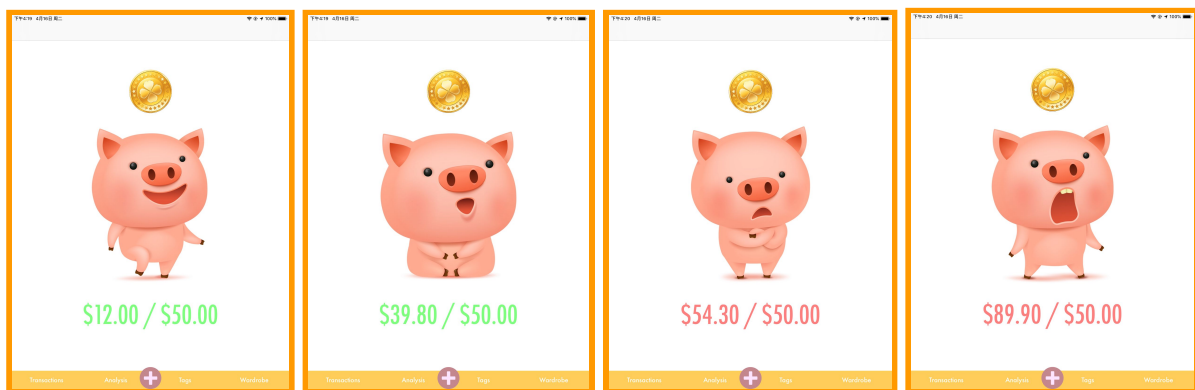
Note: We assume that the user has granted Bacon permission to use location services and send notifications, as these are required to enjoy the complete suite of features.

Budget Overview

Users may set a monthly budget. The Budget Overview page presents users with an overview of their expenditure in the current month (i.e. whether they are underspending or overspending, and how much by). The budget resets automatically after each cycle.

To give an intuitive overview for the budget, the main page of the application features our mascot, an adorable piglet (nicknamed Chris P Bacon). The piglet could be smiling, neutral, frowning, or take on any other expressions, depicting the state of the user's budget.

For instance, if the user has significantly underspent, the piglet would smile (or express a positive emotion). On the other hand, if the user has exceeded his preset budget, the piglet would frown (or express a negative emotion).



Transaction History Overview

Users may review their past transactions. Each transaction will include details such as: transaction type (i.e. income or expenditure), amount, tags, date and time, location, description (if provided) and a picture attached to the transaction (if provided).

Net Income Trend Analysis Over Time

Users may select a time period for analysis (e.g. 1st January 2019 - 30th March 2019). Users will be presented with a line chart detailing their expenditure over the selected time period.

Expenditure Breakdown by Tags

Users may select a time period (e.g. 1st January 2019 - 30th March 2019) for which they want to analyze their expenditure by tags. When recording a transaction, users may select any number of tags to attach to the transaction. This feature presents users with visual feedback detailing the proportion of their total expenditure in each tag over the selected time period.

Expenditure Breakdown by Location

Users may view their expenditure intensity at various locations on a heatmap. This lets them detect “spending hotspots” where they have spent the most. (Plus it looks really cool!)

Powerful and Flexible Transaction Recording

A transaction encapsulates numerous data points that are almost fully customizable by the user. They include:

- The type of the transaction (i.e. income or expenditure)
- The amount of the transaction
- The tags to attach to the transaction
- Whether the transaction is one-time or recurring
- The time to be associated with the transaction (i.e. now or some other time)
- The current location
- An optional picture that can be attached to the transaction
- An optional description that can be attached to the transaction

Smart Transaction Predictions

When recording a new transaction, the application attempts to automatically auto-fill (predict) transaction details based on historical data, the user’s current location and the current time of the day. Such data points include transaction amount and transaction tags. This feature may be easily extended in the future for more powerful prediction mechanisms, e.g. through the use of machine learning.

Smart Location-Driven Prompts

Bacon is location-aware, even if in the background. When it detects that the user is at a location where he is likely to make a transaction, it sends a notification prompting the user to record a transaction. As with the previous feature (smart transaction predictions), this feature can be easily extended in the future for more accurate reminders.

User Manual

System Requirements:

Bacon runs on iPads with screen sizes 9.7" and above, running iOS 12. iPhones are not supported at the moment.

Setting Up:

The first time Bacon is run, you will be prompted to set a budget for the current month. This is a compulsory step, and you cannot proceed until you have done so! This is because Bacon requires a budget to work with. Budgets are automatically reset every month. When the month ends, you will be automatically brought to this screen to set a new budget for the coming month.

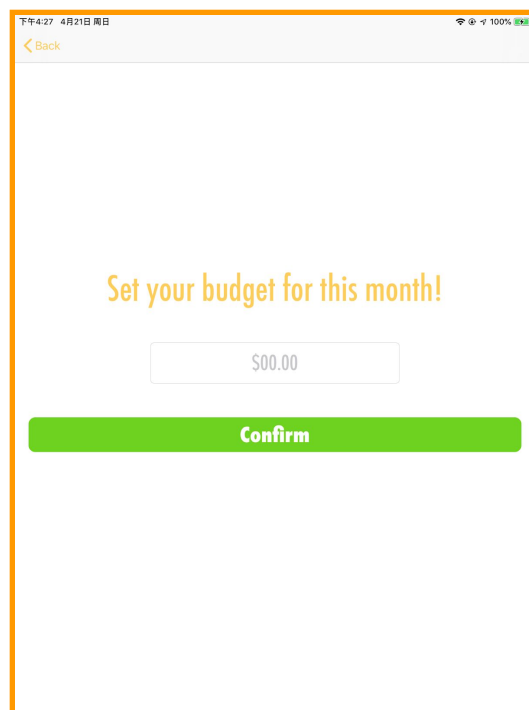


Figure 1. Set Budget Screen

Main Screen:

After setting a budget, the main screen shows the state of your budget (i.e. how much you have spent in the current period out of your budget). In addition to the number, the main page shows the mascot (an adorable piglet) whose mood changes depending on the state of your budget.

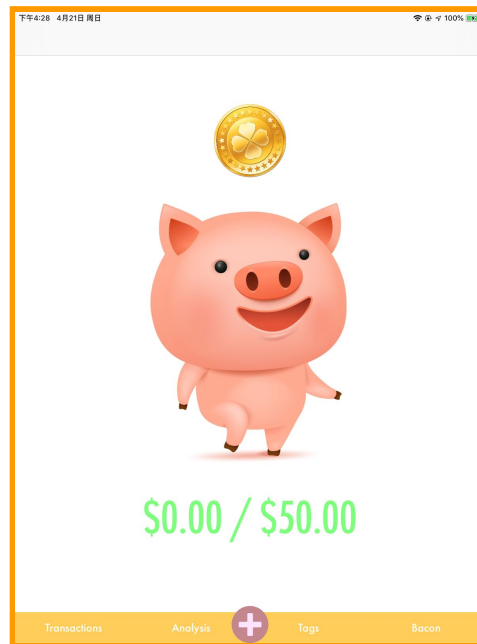


Figure 2. Main Screen

Adding a Transaction:

To add a transaction, click on the + button at the center of the bottom of the screen. Alternatively, you may swipe up the floating coin to add an expenditure transaction or swipe down to add an income transaction. You will be presented with the Add Transaction screen.

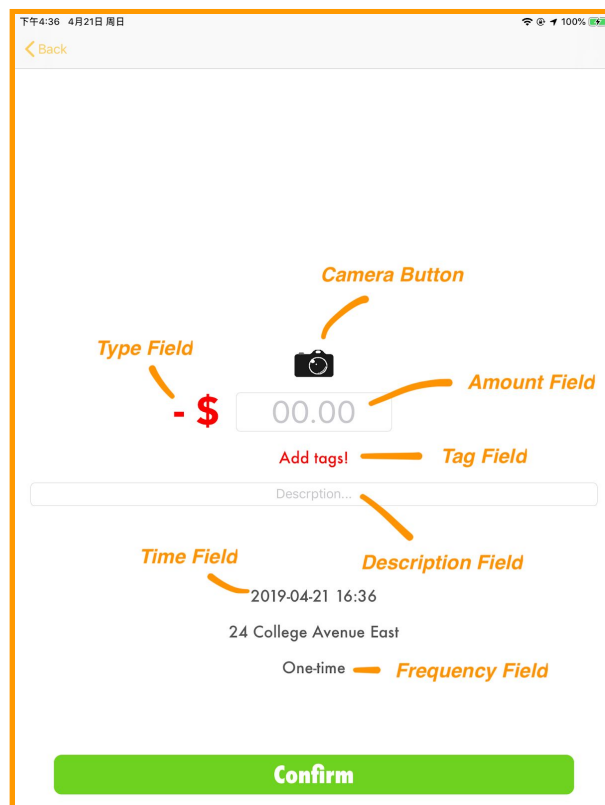



Figure 3. Add Transaction Screen (Annotated)

- To edit the type of the transaction, press the *Type Field* to toggle between expenditure and income, where a red “-” indicates expenditure and a green “+” indicates income;
- To edit the amount of the transaction, press the *Amount Field* to type in a decimal amount which is greater than 0, finishing which, press the hide key  on the keyboard to hide the keyboard;
- To add tags for the transaction, press the *Tag Field* to enter the Tag Selection screen. Press any tag name to select the tag, and press again if you wish to unselect. If you wish to add a new tag for the transaction, press the + button at the appropriate level. Press *Confirm* to confirm your choices of the tags;

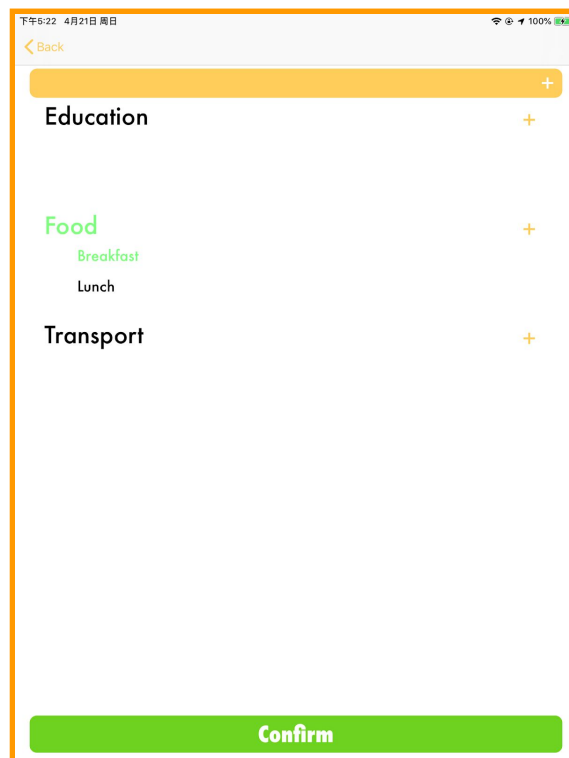



Figure 4. Tag Selection Sreen

- To edit the description of the transaction, press the *Description Field* to type in your description, finishing which, press the hide key  on the keyboard to hide the keyboard;
- To edit the time of the transaction, press the *Time Field* to enter the Calendar screen to select the date and time for the transaction. Swipe left and right to see dates in other months and press any past or future date you want to select. Press *Confirm* to confirm your choice of date and time;

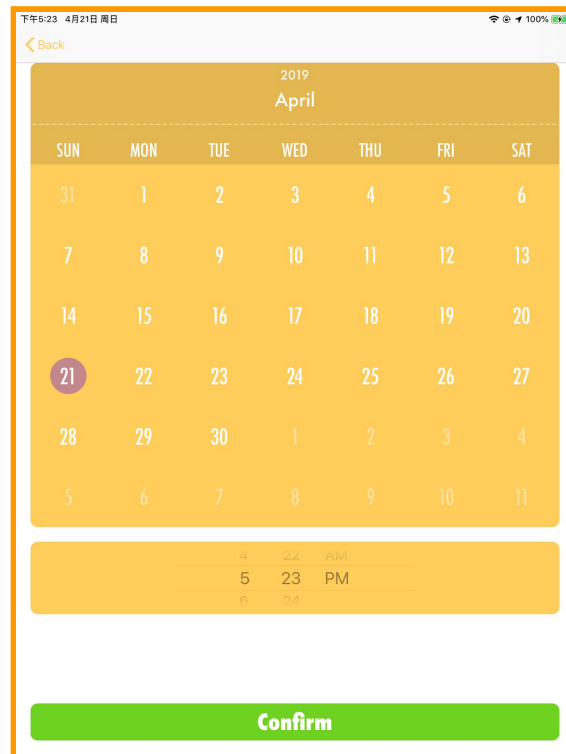


Figure 5. Calendar Screen

- To edit the frequency of the transaction, press the *Frequency Field* to toggle among one-time, daily, weekly (i.e. every 7 days), monthly (i.e. every 30 days) and yearly (i.e. every 365 days). If you wish to record a recurring transaction, provide an integer number for how many times you want this recurring transaction to repeat;
- To attach a picture for the transaction, press the *Camera Button* to take a picture from your device.

You may notice that certain details such as the amount and tags of the transaction have been automatically and conveniently pre-filled. This is because Bacon uses a smart prediction mechanism to predict your next transaction based on your current location, current time of the day and your transaction histories to guess what you are about to transact next!

Viewing Past Transactions:

To view the transactions, press the *Transactions* tab from the Main Screen. You will be presented with the Transactions Screen.

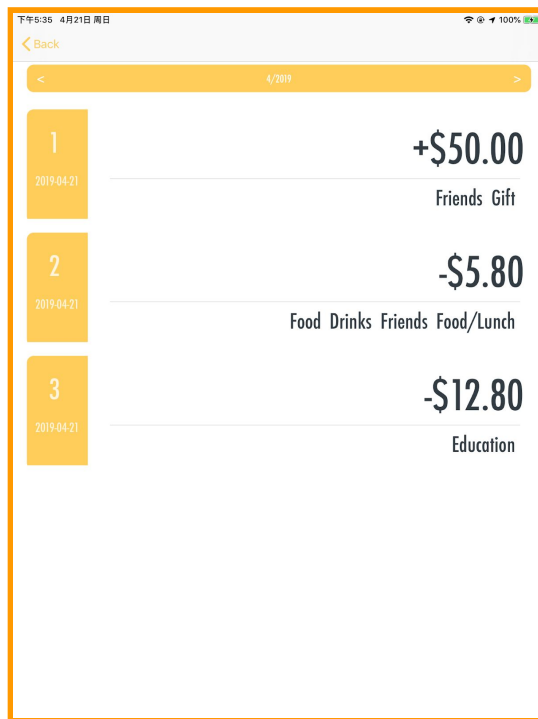


Figure 6. Transactions Screen

You will see a list of transactions you have previously recorded for the current month. To see transactions of other months, press < or > in the top bar for the previous or the next month. By default, the amount, tags and date of each transaction will be shown. To see more details of a particular transaction, press the transaction to expand the chosen transaction entry.

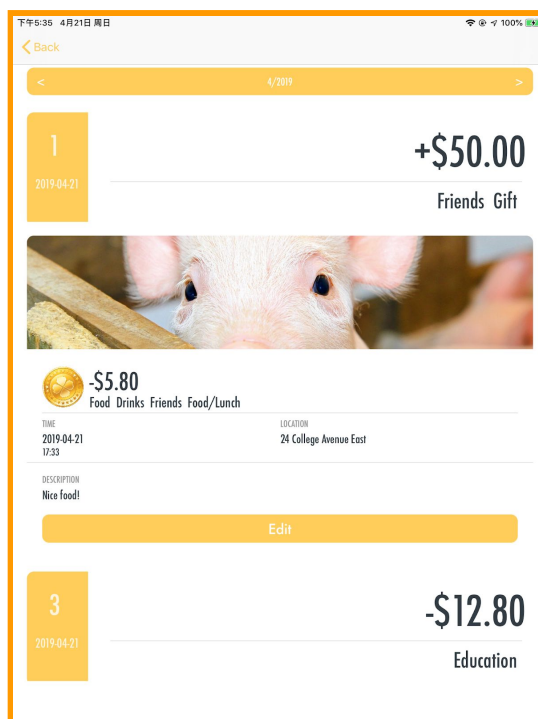


Figure 7. Transactions Screen with the Second Transaction Entry Expanded

The picture you attached with the transaction may not be displayed fully from the expanded transaction entry. Fret not! Press the picture to see the picture in full.



Figure 8. Showing the Picture of a Transaction

Editing a Transaction:

To edit a transaction, press the *Edit Button* from the expanded transaction entry explained above. You will be presented with a screen similar to the Add Transaction Screen shown above. Edit any number of fields you wish to change for the transaction.

However, there are certain restrictions about which fields are editable depending on whether you are editing a one-time transaction or one belonging to a series of a recurring transaction. The rules are as follows:

1. A one-time transaction cannot be changed to become recurring. If you wish to do this, please add a new recurring transaction with the desired details;
2. A recurring transaction cannot be changed to become one-time. However, you can change the recurring interval (e.g. weekly, monthly etc.) and the number of times you wish to repeat the transaction;
3. The time associated with a recurring transaction cannot be changed.

Note that editing any transaction belonging to the same recurring series is equivalent.

Deleting a Transaction:

To delete a transaction, swipe left at the transaction entry you wish to delete. If it is a one-time transaction, it will be deleted right away. If it is one belonging to a recurring series, you will be prompted with the following and please choose the appropriate option.

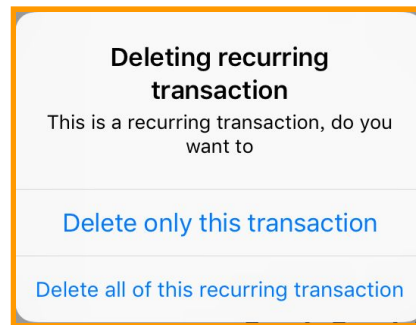


Figure 9. Prompt when Deleteing a Recurring Transaction

Analysing Monthly Net Income Trend:

To analyse your monthly net income trend, press the *Analysis* tab from the Main Screen. Choose the time period you wish to analyse by pressing the two buttons at the bottom of the screen, and a trend graph will be generated right away.

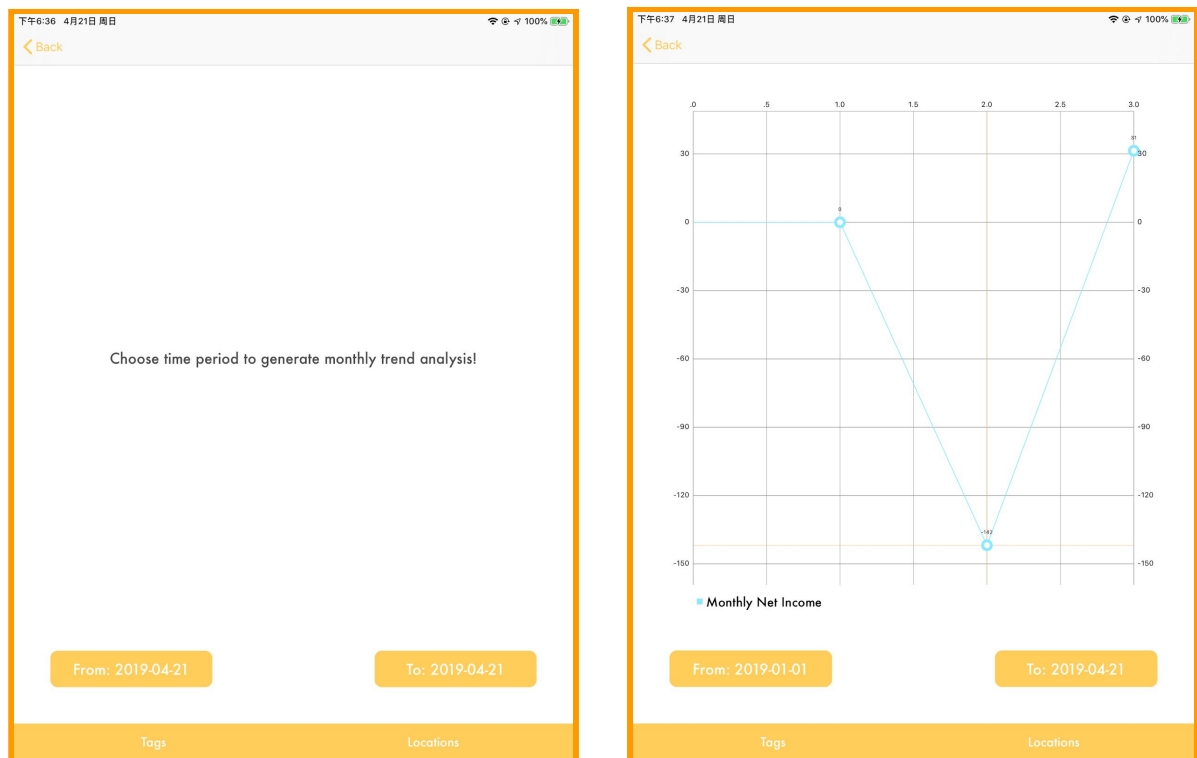


Figure 10. Trend Analysis Screen before and after Choosing Time Period

Analysing Expenditure Breakdown by Tag:

To analyse your expenditure breakdown by tag, press the *Tags* tab from the Analysis Screen. Choose the time period and the tags you wish to analyse by pressing the three buttons at the bottom of the screen, and a bar graph will be generated right away.



Figure 11. Tag Analysis Screen before and after Choosing Time Period and Tags

Analysing Expenditure Breakdown by Location:

To analyse your expenditure breakdown by location, press the *Locations* tab from the Analysis Screen. Choose the time period you wish to analyse by pressing the two buttons at the centre of the screen, and press the *Confirm Button* to continue to the Heatmap Screen. For places where a large amount of expenditure has been recorded, it will appear red on the heatmap.

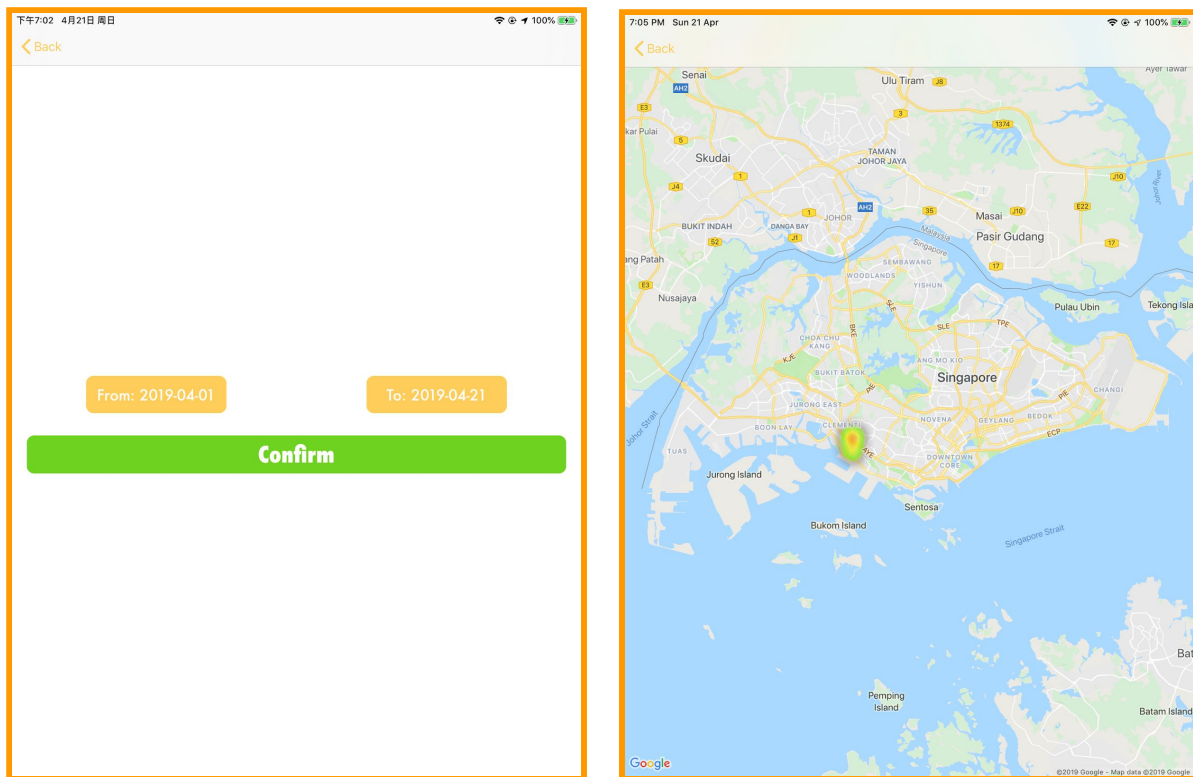


Figure 12. Location Analysis Screen before and after Choosing Time Period

Editing a Tag:

To edit a tag, press the *Tags* tab from the Main Screen. Press the tag and you will be presented with the following prompt window. Type in the new name to continue.

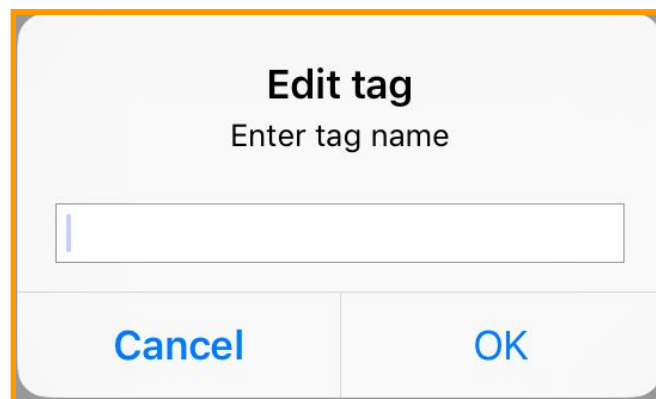


Figure 13. Prompt Window For Editing a Tag

Deleting a Tag:

To delete a tag, press the *Tags* tab from the Main Screen. Swipe left at the tag entry you wish to delete. If it is a top-level tag, itself and all its sub-tags will be deleted. If it is a sub-tag, only itself will be deleted. All transactions previously tagged with the deleted tag will no longer be associated with the deleted tag.

Resetting the Budget:

To reset the budget, press the Budget tab from the Main Screen. Alternatively, you can also press on the budget status shown in the Main Screen to enter the Set Budget Screen explained at the start of the section.

Performance

The application is not CPU/GPU intensive. However, it may potentially be memory-intensive, especially if the user has a large number of transactions that are loaded into memory.

The decision between implementing a transaction as a class or struct is ultimately one of whether a transaction is best represented as a reference or value type. On this end, we thought that a transaction was better modeled as a reference type than a value type.

Transactions have *identities* - 2 transactions at the same date, time, location, and of the same amount, category, type, etc. are nevertheless 2 distinct transactions.

Transactions come in many different varieties: e.g. one-time vs recurring transactions, expenditure-type vs income-type transactions, etc. At first blush, this seems like a classic use-case of classical inheritance (both one-time transactions and recurring transactions “*is a*” transaction). However, upon deeper consideration, inheritance is undesirable, due to the large number of permutations possible (e.g. one-time vs recurring transactions + expenditure-type vs income-type + 12 transaction categories results in 48 different permutations). Instead, we decided to use composition - a transaction is composed of a transaction-type, a transaction-frequency, a transaction-category, etc. Therefore, even though we are representing transactions as classes, we will not be using inheritance to model the different types of transactions.

The abstraction function of a transaction is straightforward: the abstract transaction type is simply the aggregate of its concrete representation properties.

Because transactions are implemented as mutable reference types, we require a `checkRep()` method to ensure that any mutation to its properties results in a valid representation state. For instance, the “amount” property cannot be negative.

Where there is an attempt to initialize a transaction with invalid arguments, we have the option to either fail (through a failable initializer) or throw an error. We opted to throw an error (`InitializationError`) for the following reasons:

- 1) It makes the failure more explicit by forcing the caller to wrap the initialization statement in a try-catch statement and handle any errors thrown during initialization
- 2) More importantly, we are able to provide an error message detailing the exact reason that initialization failed, which will be subsequently feedbacked to the user for correction

Observing Transactions: Modifications

Since transactions are represented by `Transaction` instances, transaction properties (e.g. amount, category, date/time) should be modifiable. Modifications to transaction details should also be persistent (i.e. written to disk).

However, we could not allow `Transaction` properties to be directly mutable, because we needed to guard against invalid input, and there is no way to prevent the setting of a new (invalid) value using Swift’s `willSet` or `didSet` observer functions.

At this juncture, we had 2 alternatives: 1) we could use computed properties, which allows us to handle invalid values; or 2) we could use a `.edit()` instance method.

We opted to use the latter. Even though it was less “elegant”, it allowed us to edit multiple properties in one operation, which was a huge advantage. We can now notify observers of all the changes in one notification, rather than n notifications (for n changes).

To accomplish this, we implemented a generic Observer-Observable framework for use in our project. We conform the Transaction class to Observable, and TransactionManager as an Observer. Transactions are created/loaded through TransactionManager, which registers itself as an Observer of each Transaction instance before returning the instance.

We also implemented an instance method edit() which accepts 10 arguments (8 optional arguments for each Transaction property, 1 success callback, and 1 failure callback). The 8 property arguments are optional types so that as many arguments may be passed as needed to be modified. The callback arguments ensure that the caller may be appropriately notified of the outcome of the operation.

When any of the Transaction properties is modified through the edit() method, TransactionManager calls StorageManager to write the new transaction details to disk.

Observing Transactions: Deletion

A transaction can also be deleted (e.g. if the user spots a duplicate transaction). To support transaction deletion through the object-mutation paradigm, we modified the Transaction class to support a .delete() method, which accepts 2 callbacks to be called in the event of deletion success and failure respectively.

When the user deletes a transaction, the Controller calls the .delete() method of the transaction instance. TransactionManager, which observes the transaction, is notified and calls StorageManager to attempt to delete the transaction record from disk. If it succeeds, the success callback is fired. Otherwise, if the operation fails for whatever reason, the failure callback is fired. The callbacks allow for feedback to be presented to the user, without breaking abstraction barriers or excessive coupling between the components.

Module structure

CoreLogic: Singleton?

The CoreLogic component is the “brain” of the application. It is responsible for application logic and coordination between its different subcomponents. It is also the proxy between its subcomponents and the view controllers.

Therefore, we recognize the argument for implementing it as a singleton pattern, considering that there should only really be a single instance of CoreLogic throughout the application. This allows any ViewController to access the singleton instance, without having to pass the instance between view controllers.

At the same time, we also acknowledge the downside to a singleton design pattern. Singletons reduce testability, introduce global state, and prevent true isolation of dependent classes.

Since we have decided to implement StorageCouchDB (see next section) as a singleton, we weighed the benefits and downsides to a singleton pattern approach for CoreLogic, and decided not to implement it as a singleton class after all.

StorageCouchbaseDB: Singleton

The StorageCouchbaseDB class is the low-level class that communicates with the database. We decided to implement it as a singleton, since we never want to create multiple connections with the database. At any one time, we only want a single connection with it.

Separation of TransactionManager, BudgetManager, and PredictionManager

Our Model (of the MVC architectural pattern) is represented by TransactionManager, BudgetManager, and PredictionManager. StorageManager exists as an abstraction class (proxy) over the specific database implementation.

TransactionManager, BudgetManager, and PredictionManager serve as the proxies between the application and StorageManager for working with their respective models (transactions, budget, and predictions). StorageManager serves as the proxy between these modules and the database. We thought this distinction between the 3 classes was beneficial.

Firstly, it allows us to practise separation of concerns. Each class has its own responsibilities. This allows for better maintainability and testability. It also allows us to extend features in the future more easily.

Furthermore, this has the additional benefit of allowing for the mechanism of storage to be changed in the future, without affecting the Controller (in MVC). For instance, if we wish to switch to a different database, we could simply swap out StorageManager.

To work with these 3 classes, we applied the facade design pattern to create a CoreLogic component. The Controller (View Controller) communicates solely with CoreLogic, which in turn relays (delegates) method calls to the appropriate Model components. This reduces coupling within our application which improves maintainability and extensibility.

Memory vs disk

We faced a design choice of whether to load all transactions into memory on app startup or keep them on disk in a file-based database. If we choose the first option, we are effectively replicating the database in memory to work with for the usage session, and only rewriting it onto the disk-based database at the end of the usage session for persistency.

The benefit of loading transactions into memory is improved performance, as read/write operations are a lot faster from/to memory than disk. However, the downside to doing so is that this design is not scalable, as the memory usage grows linearly with respect to the number of transactions. Furthermore, we most likely will not require all of the stored transactions in a usage session, which translates to wasted time and memory expended if we were to load all the data whenever we start the app.

In the end, we decided to do lazy-loading which is to keep transactions on disk, and load them as and when we require them. Suppose a use case where the user opens the app just to check the budget overview for the current month, the benefit of lazy-loading is apparent here as loading transactions from last year is redundant.

As the popular saying goes, “premature optimization is the root of all evil”. Without concrete evidence (e.g. from a performance profiler) to suggest a significant performance penalty in reading/writing from/to the disk, we opted to work with the safer and more scalable option of lazy-loading.

We do acknowledge that this may turn out to be an ill-judged decision, especially if we need to load a large amount of transactions from disk and suffer a performance penalty consequently. However, we believe that during normal usage, we should not be dealing with such a large amount of transactions as to suffer any significant or noticeable performance penalty.

Ready interface for advanced prediction mechanisms

For the Transaction Prediction feature, currently we have defined a fixed set of rules for the prediction mechanism. For example, to predict tags, we loop through all similar past transactions happening at similar locations and similar time of the day to take the top three most frequently used tags. While this method does work to some extent, we acknowledge that it is not the “smartest” way. With the availability of many Machine Learning models and frameworks on the market, it may have been better for us to deploy some advanced learning algorithms to do the prediction. However, since algorithms are not the focus of this project and the overhead of learning a new library/framework is high, we decided not to use a real Machine Learning model for the prediction.

Nevertheless, having in mind the idea above, we have built an interface that makes integration with an actual Machine Learning model easy. For example, we have provided the PredictionGenerator Interface where a method `predict()`, which takes in a time, a location and a list of past transactions and returns a Prediction result, is defined. It corresponds to

the interface of many Machine Learning models, where the list of transactions can be used to train and tune the hyper-parameters of the model while the time and location are used for the actual prediction by the model. By providing this abstraction over the underlying prediction generation, it is easy for us to swap out the actual generator to such a Machine Learning model in the future. On top of this, we have also set up a storage mechanism for accepted predictions (i.e. we save the generated Prediction result if the user actually accepts the given Prediction), hoping this will help better train the model if we use one in the future.

Database choice

One of the most important decisions we had to make was the choice of database. We wanted an embedded database that runs locally on-device, without requiring services or network connections. This is preferred over simply storing JSON files, as database libraries allow efficient storing and querying.

We considered the following candidates: SQLite, MongoDB Mobile, and Couchbase Lite. We opted to use Couchbase Lite in the end, for the following reasons.

We considered SQLite as one of the candidates as it is an extremely popular database that is proven and tested. Its popularity also means we would easily be able to find documentation and support online. However, since our transaction struct conform to Codable, Swift automatically encodes/decodes them to/from JSON data. Thus, we thought it would be most appropriate to use a NoSQL database (which rules out SQLite), as it dispenses with the need to convert between SQL database rows and transaction instances in the application.

MongoDB Mobile was an attractive option as there is a third-party library (MongoKitten) which supports the Encodable and Decodable (Codable) protocols by providing the BSONEncoder and BSONDecoder type, with the difference being that BSONEncoder produces instances of Document instead of Data. MongoDB and Couchbase both support saving and loading of Document type instead of Data. This would make the implementation of saving and loading very straightforward. However, MongoKitten is still in beta and is not currently supported.

Between MongoDB Mobile and Couchbase Lite, Couchbase Lite provides programming syntax that's familiar to mobile and server developers alike. Couchbase provides a SQL-like query language along with familiar JOIN capabilities and powerful full-text search features, whereas MongoDB has a proprietary API which would require a learning curve. Eventually, we decided on Couchbase Lite, as it is a mature market-tested platform whilst MongoDB Mobile is a very young product, that is still currently in beta.

Charting

There are a ton of charting libraries available for Swift. As none of us have had any personal experience using any of those, we decided to opt for the safe option by using a library that is popular, widely-used, and well-maintained. We gauged these factors through GitHub

stars/watches, commit frequency, and last-commit date. We decided to use Charts (<https://github.com/danielgindi/Charts>) as it provides the features we need, and scores very highly on the previously-mentioned factors.

Testing

Testing Strategy

Overview

We use a combination of blackbox and glassbox testing for testing various components of our application. In designing our application, we applied protocol-oriented programming and composition as much as possible, resulting in small, discrete components that are highly testable.

We created a common pool of test data to be used across our different test suites. In generating the test data, we use a mixture of valid and invalid input. The motivation behind doing so is to maintain a single source of test data that can be modified (and improved) in the future, with the improvement automatically applying to the various test suites.

Other than test suites and test cases, we also use run-time assertions in some components that are more complicated to ensure that conditions and invariants are maintained.

We set up continuous integration with Travis CI and protected our main (master) branch by requiring all pull requests builds to pass. This guards against the accidental introduction of regressions in application development.

Utility Objects

Utility objects are grouped under “Commons”, and refer to classes, structs, and protocols that are not models (under the MVC architectural pattern), but are instead common definitions that serve a utility purpose for use throughout the application. An example of a utility object is CodableCLLocation (a wrapper class around Swift’s CLLocation object that conforms to the Codable protocol, since CLLocation does not).

Utility objects have their own unit tests to ensure the correctness of their respective functions. For instance, CodableCLLocation is tested to verify that its instance may be decoded and encoded successfully without any loss/corruption of information; the Observer pattern (which comes with default implementation) is tested to verify that observers are successfully notified when an Observable is mutated.

Utility objects are tested using glassbox testing, and tests are written by the implementer who would naturally be most familiar with the implementation.

Models

Models also have their own unit tests to check the algorithmic correctness of their public functions and computed properties (API). We use glassbox testing and various heuristics (e.g. equivalence partitioning and boundary value analysis) to ensure that functions: 1) return the correct output given some inputs; 2) handle edge cases correctly; 3) throw exceptions where appropriate. This includes testing branches in the code, such as if and guard statements.

Controller

Our controller component of the MVC pattern can be split into 2 sub-categories: view controllers and our CoreLogic component.

View controllers are tested manually using blackbox testing, due to the complexity of writing UI tests.

Our CoreLogic component has unit tests to verify its correctness. It is not as rigorously tested as our utility objects and models. However, since our CoreLogic component acts as the coordinator between utility and model objects, we feel it is more important to focus on unit testing utility and model objects. CoreLogic unit tests are essentially integration tests of its sub-components.

Concluding Remarks

Given our emphasis on unit testing, we expect few (if any) bugs with functionality and logic of our components. We also don't expect bugs in our views (UI), as they don't contain a lot of logic.

We might encounter bugs pertaining to the integration of our different components, given that we are less focused on integration tests. However, as we have yet to encounter any from manual testing, we don't believe them to be a significant problem (if at all).

Testing Results

To summarize what has been stated in the earlier section, we presently have rigorous unit tests for our components, and less rigorous integration tests for our controller.

Core, fundamental components and modules (e.g. the Transaction class, TagManager module, and Storage module) have been comprehensively glassbox-tested with numerous test cases testing various aspects of them: algorithmic correctness, edge cases, invalid input, etc. We are confident that these work as expected.

We could have included more integration tests for our controller component, to further reduce the possibility of integration bugs. However, we have extensively manual-tested our application, and to date, we have not encountered any.

Reflection

Evaluation

On the whole, we are very proud of the architecture of our product. As can be seen from our class diagram, we have divided our product into numerous components and effectively applied the facade pattern (through the CoreLogic component) to minimize undesirable coupling between components. We also relied heavily on protocol-oriented programming to practice design by contract. As a result, we were able to apply the SOLID design principles extensively.

One of the implementation techniques that we are especially proud of is the generic Observer-Observable implementation that allows any object to observe another object, where both objects may be of any type. Although we only applied it to our Transaction class, the implementation of a generic Observer-Observable pattern allows us to easily extend it to more classes in the future, should the need arise (in the hypothetical scenario where we need to maintain the software).

Having our application separated into multiple subcomponents with clearly-defined interfaces was one of the greatest successes of our project. It allows us to work concurrently on different parts of the application without worrying about integration problems, since we have clearly-defined specifications in our interfaces. Although this meant that initial progress is slow, as we needed to discuss and finalize the specifications of the interfaces, we felt that the trade-off is well worth it, as it allowed us to work more efficiently subsequently and minimized the presence of integration bugs.

One of the problems we discovered late into the project was that we were operating under a mistaken assumption that read/write operations with a local file-based database would have negligible performance costs. When attempting to save/load a large number of transactions (especially if they have images attached to them), there is a perceptible lag. This was because read/write operations were performed synchronously.

Finally, another problem that we faced was ensuring consistency in coding style and maintaining code quality. While we used swiftlint to catch major problems, and enforced a system of peer code review, we could have done more, such as through automated code review in CI.

Lessons

Previously, we mentioned that we have concerns about falling behind schedule. We thought that we spent too much time perfecting (or working towards perfection) our fundamental base classes, structs, modules, and components. However, in hindsight, we felt that we made the right decision, as we did not encounter any bugs/quirks in our subsequent

development, which was extremely fortunate. That said, a balance must always be struck, and perhaps we could have expedited that process slightly, which would afford us more time in the later stages of development to develop more features.

Secondly, we would have implemented read/write operations differently, to mitigate performance issues. Some alternative implementations that we would use are to take advantage of multithreading (through GCD), or rewrite read/write functions as asynchronous ones with success and failure callbacks.

Furthermore, on a side note, we could have mitigated performance issues without rewriting our read/write mechanisms by compressing image files initially to a significantly reduced file size.

Known bugs and limitations

1. When attempting to save a large number (about 200) of recurring transactions with pictures attached, the application runs out of memory (runtime error). This is because we are working with uncompressed pictures which can be several MBs in size. In hindsight, we should really only be saving one copy of the picture, and storing references to it in all the other recurring transaction instances.
2. This is not exactly a bug, but write operations are not guaranteed to be atomic, which is a limitation.

Appendix

Formats

No formats are used in our application.

Module specifications

Transaction

This encapsulates various properties that together define a transaction instance.

Properties:

- date: Date
- type: TransactionType
- frequency: TransactionFrequency
- tags: Set<Tag>
- amount: Decimal
- description: String
- image: CodableUIImage?
- location: CodableCLLocation?
- recurringID: UUID?

TransactionFrequency

This represents the frequency of a transaction (i.e. one-time or recurring, how often it repeats, and how many times).

Properties:

- nature: TransactionFrequencyNature
- interval: TransactionFrequencyInterval?
- repeats: Ints?

TagManagerInterface

This is an interface that TagManager conforms to.

Properties:

- getTag(for value: String, of parentValue: String?) throws -> Tag: Returns a Tag of the provided values.
- func addChildTag(_ child: String, to parent: String) throws -> Tag: Adds a new child Tag to a parent Tag and returns it.

- func addParentTag(_ parent: String) throws -> Tag: Adds a new parent Tag and returns it.
- func removeChildTag(_ child: String, from parent: String) throws -> [Tag]: Removes a child Tag from a parent Tag.
- func removeParentTag(_ parent: String) throws -> [Tag]: Removes a parent Tag. All of its children Tags will be removed too.
- var tags: [Tag: [Tag]] { get }: Contains all Tags in a dictionary mapping parent Tags to sorted arrays of their children Tags.
- var parentTags: [Tag] { get }: Contains a sorted array of all existing parent Tags.
- func getChildrenTags(of parent: String) throws -> [Tag]: Returns a sorted array of the children Tags of a parent Tag.
- func isChildTag(_ child: String, of parent: String) -> Bool: Checks whether a child Tag exists.
- func isParentTag(_ parent: String) -> Bool: Checks whether a parent Tag with the provided value exists.
- func renameTag(_ oldValue: String, to newValue: String, of parent: String?) throws -> Tag: Renames a Tag and returns it.
- func clearTags(): Clears all Tags.

Tag

Represents a tag to attach to Transaction instances. A tag may be a parent tag (top-level) for a child tag.

- internalValue: String
- parentInternalValue: String?
- value: String
- parentValue: String?

StorageManagerInterface

This is an interface that StorageManager conforms to.

- func getNumberOfTransactionsInDatabase() -> Double: Returns the number of transactions in the database.
- func clearTransactionDatabase() throws: Clears the database of transactions.
- func saveTransaction(_ transaction: Transaction) throws: Saves a Transaction to the database.
- func deleteTransaction(_ transaction: Transaction) throws: Deletes a Transaction. This should only be called on Transactions that are loaded out from the database.
- func deleteAllRecurringInstances(of transaction: Transaction) throws: Deletes all Transaction instances with the same recurring ID.
- func updateTransaction(_ transaction: Transaction) throws: Updates a Transaction. This should only be called on Transactions that are loaded out from the database.
- func loadAllTransactions() throws -> [Transaction]: Loads all the Transactions in the database. The caller is responsible for ensuring that doing so will not result in a memory warning or error.

- func loadTransactions(limit: Int) throws -> [Transaction]: Loads a collection of Transaction objects.
- func loadTransactions(after date: Date, limit: Int) throws -> [Transaction]: Loads a collection of Transaction objects after the specified date.
- func loadTransactions(before date: Date, limit: Int) throws -> [Transaction]: Loads a collection of Transaction objects before the date specified.
- func loadTransactions(from fromDate: Date, to toDate: Date) throws -> [Transaction]: Loads a collection of Transaction objects between the specified dates (inclusive).
- func loadTransactions(ofType type: TransactionType, limit: Int) throws -> [Transaction]: Loads a collection of Transaction objects of the specified type.
- func loadTransactions(ofTag tag: Tag) throws -> [Transaction]: Loads a collection of Transaction objects with the tags specified.
- func loadFirstRecurringInstance(of transaction: Transaction) throws -> Transaction: Loads and returns the first Transaction instance of a recurring Transaction object.
- func getNumberOfBudgetsInDatabase() -> Double: Returns the number of budgets in the database.
- func clearBudgetDatabase() throws: Clears the database of the budget.
- func saveBudget(_ budget: Budget) throws: Saves a budget to the database. There can only be at most 1 budget existing in the database. Therefore, this will overwrite any existing budget data.
- func loadBudget() throws -> Budget: Returns the set budget.
- func deleteTagFromTransactions(_ tag: Tag) throws: deleteTagFromTransactions will remove the specified tag from all transactions associated with it.

LocationPrompt

This is a class that exposes one public static method.

Properties:

- static func shouldPromptUser(currentLocation: CLLocation, decisionHandler: @escaping (Bool) -> Void): Decides if the user should receive a prompt to record a transaction.

ApiHandlerProtocol

API handlers send requests to and interpret responses from specific remote resources to determine if a user should receive a prompt based on the current location.

Properties:

- func sendRequest(currentLocation: CLLocation, decisionHandler: @escaping (Bool) -> Void): Sends a request to this ApiHandler's API endpoint, interprets the response (or handles a failure), and calls `decisionHandler` with the decision (`true` to prompt and `false` otherwise).

Test cases

Unit testing - Black-box testing

StorageManager # Transactions

- `getNumberOfTransactionsInDatabase()`
 - Test on empty database
 - Should return 0
 - Test on non-empty database
 - Should return the correct number of transactions in database
- `clearTransactionDatabase()`
 - Test on empty database
 - Should not throw
 - Should do nothing
 - Test on non-empty database
 - Should not throw
 - Should clear all transactions
- `saveTransaction(transaction)`
 - Should save transaction in database
- `deleteTransaction(transaction)`
 - Test a invalid transaction
 - Should throw error
 - Test a valid transaction
 - Should delete transaction in database
- `deleteAllRecurringInstances(transaction)`
 - Test a .oneTime transaction
 - Should throw error
 - Test a recurring transaction
 - Should delete all recurring instances of the transaction (i.e. transactions with the same recurring id should be removed)
- `updateTransaction(transaction)`
 - Test a invalid transaction (does not exist in database)
 - Should throw error
 - Test a valid transaction (exist in database)
 - Should save the updated transaction into database
- `loadAllTransactions()`
 - Test empty database
 - Should return empty array
 - Test non-empty database
 - Should return all the transactions saved in the database in reverse chronological order
- `loadTransactions(limit)`
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)

- Should throw an error
- loadTransactions(after, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions after *after* date in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions after *after* date in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadTransactions(before, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions before *before* date in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions before *before* date in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadTransactions(from, to)
 - Test a valid time range (i.e. *to* is equal to *after* from)
 - Should return all transactions within the specified period in reverse chronological order in an array
 - Test an invalid time range (i.e. *to* is before *from*)
 - Should throw an error
- loadTransactions(ofType, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions of type *ofType* in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions of type *ofType* in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadTransactions(ofTag, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions of tag *ofTag* in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions of tag *ofTag* in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadFirstRecurringInstance(transaction)
 - Test a .oneTime transaction
 - Should throw error

- Test a .recurring transaction
 - Should return the first occurrence of the recurring transaction

StorageManager # Budget

- getNumberOfBudgetsInDatabase()
 - Test on empty database
 - Should return 0
 - Test on non-empty database
 - Should return 1 (There is always only 1 budget in the database)
- clearBudgetDatabase()
 - Test on empty database
 - Should not throw
 - Should do nothing
 - Test on non-empty database
 - Should not throw
 - Should clear the budget
- saveBudget(budget)
 - Test on empty database
 - Should save budget
 - Test on non-empty database
 - Should overwrite previous budget
- loadBudget()
 - Test on empty database
 - Should throw error
 - Test on non-empty database
 - Should return the budget saved in the database

StorageManager # Tag

- deleteTagFromTransactions(tag)
 - Test invalid tag
 - Should throw error
 - Test valid tag
 - Should remove the tag from all transactions that contains it in the database.

StorageManager # Prediction

- getNumberOfPredictionsInDatabase()
 - Test on empty database
 - Should return 0
 - Test on non-empty database
 - Should return the number of predictions saved in the database
- clearPredictionDatabase()
 - Test on empty database
 - Should not throw
 - Should do nothing
 - Test on non-empty database

- Should not throw
 - Should clear all predictions
- savePrediction(prediction)
 - Should save prediction into database
- loadAllPredictions()
 - Test on empty database
 - Should return empty array
 - Test on non-empty database
 - Should return an array with all the predictions saved in the database
- loadPredictions(limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of predictions in database}$)
 - Should return the latest *limit* number of predictions in reverse chronological order in an array
 - Test a valid limit ($\text{limit} > \text{number of predictions in database}$)
 - Should return all predictions in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error

TransactionManager

- getNumberOfTransactionsInDatabase()
 - Test on empty database
 - Should return 0
 - Test on non-empty database
 - Should return the correct number of transactions in database
- clearTransactionDatabase()
 - Test on empty database
 - Should not throw
 - Should do nothing
 - Test on non-empty database
 - Should not throw
 - Should clear all transactions
- saveTransaction(transaction)
 - Should save transaction in database
- updateRecurringTransaction(transaction)
 - Test a .oneTime transaction
 - Should throw error
 - Test a .recurring transaction
 - The updated recurring transaction should have its changes updated to all recurring instances.
- deleteTagFromTransactions(tag)
 - Test invalid tag
 - Should throw error
 - Test valid tag
 - Should remove the tag from all transactions that contain it in the database.
- deleteAllRecurringInstances(transaction)

- Test a .oneTime transaction
 - Should throw error
- Test a recurring transaction
 - Should delete all recurring instances of the transaction (i.e. transactions with the same recurring id should be removed)
- loadTransactions(limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadTransactions(after, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions after *after* date in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions after *after* date in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadTransactions(before, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions before *before* date in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions before *before* date in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error
- loadTransactions(from, to)
 - Test a valid time range (i.e. to is equal to after from)
 - Should return all transactions within the specified period in reverse chronological order in an array
 - Test an invalid time range (i.e. to is before from)
 - Should throw an error
- loadTransactions(ofType, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions of type *ofType* in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions of type *ofType* in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)

- Should throw an error
- loadTransactions(ofTag, limit)
 - Test a valid limit (i.e. $0 < \text{limit} \leq \text{number of transactions in database}$)
 - Should return the latest *limit* number of transactions of tag *ofTag* in reverse chronological order in an array
 - Test a valid limit (i.e. $\text{limit} > \text{number of transactions in database}$)
 - Should return all transactions of tag *ofTag* in reverse chronological order in an array
 - Test an invalid limit (i.e. $\text{limit} < 0$)
 - Should throw an error

BudgetManager

- saveBudget(budget)
 - Test on empty database
 - Should save budget
 - Test on non-empty database
 - Should overwrite previous budget
- loadBudget()
 - Test on empty database
 - Should throw error
 - Test on non-empty database
 - Should return the budget saved in the database
- deleteBudget()
 - Test on empty database
 - Should not throw
 - Should do nothing
 - Test on non-empty database
 - Should not throw
 - Should clear the budget

PredictionManager

- savePrediction(prediction)
 - Should save transaction in database

CoreLogic

- getTotalTransactionsRecorded() -> Double
 - Test on empty database
 - Should return 0
 - Test on non-empty database
 - Should return the correct number of transactions in database
- clearAllTransactions()
 - Test on empty database
 - Should not throw
 - Should do nothing
 - Test on non-empty database

- Should not throw
 - Should clear all transactions
- recordTransaction(date, type, frequency, category, amount, location, images)
 - Test valid arguments
 - Should record a transaction of date, type, frequency, category, amount, location, and images and return the Transaction object.
 - Test invalid arguments
 - Test invalid frequency
 - Should throw InitializationError
 - Test invalid amount (i.e. amount <= 0)
 - Should throw InitializationError
- deleteAllRecurringInstances(transaction)
 - Test a .oneTime transaction
 - Should throw error
 - Test a recurring transaction
 - Should delete all recurring instances of the transaction (i.e. transactions with the same recurring id should be removed)
- loadTransaction(forMonth, inYear)
 - Test valid arguments
 - Should return an array of transactions for the specified month and year
 - Test invalid arguments
 - Test invalid month (i.e. month <= 0 or month > 12)
 - Should throw an error
- getBreakdownByTag(from, to, tags)
 - Test a valid time range (i.e. 'from' does not occur after 'to')
 - Should return a dictionary mapping transaction tags to the expenditure on that category within the specified time period
 - Test an invalid time range (i.e. 'to' occurs before 'from')
 - Should return the dictionary with all values set to 0.

UI testing - Black-box testing

- Test Main Page (landing page)
 - If opened for the first time
 - Lead to Set Budget Page
 - If budget is never set
 - Lead to Set Budget Page
 - If budget is overdue
 - Lead to Set Budget Page
 - Test Pig View
 - If current spending is above 0% and below 50% of total budget
 - Shows very happy pig
 - If current spending is at least 50% and below 100% of total budget
 - Shows happy pig
 - If current spending is above 100% and below 150% of total budget

- Shows sad pig
 - If current spending is at least 150% total budget
 - Shows very sad pig
 - If current spending is equal to budget
 - Shows neutral pig
- Test Number View
 - If current spending is below budget
 - Number in green color
 - If current spending is above budget
 - Number in red color
 - If current spending is equal to budget
 - Number in beige color
- Test Coin View
 - Floating up and down (animation)
 - Swipe up
 - Lead to Add Transaction Page (with expenditure type)
 - Swipe down
 - Lead to Add Transaction Page (with income type)
- Test Add Transaction Page
 - Test Transaction Type Field
 - If entered the page from Add Button (+) in Main Page
 - Type label shows “-” to indicate expenditure
 - Type label in red to indicate expenditure
 - If entered the page from swiping up in Main Page
 - Type label shows “-” to indicate expenditure
 - Type label in red to indicate expenditure
 - If entered the page from swiping down in Main Page
 - Type label shows “+” to indicate income
 - Type label in green to indicate income
 - Test Add Tag Button
 - If current transaction type is expenditure
 - Button appears red to indicate expenditure
 - If current transaction type is income
 - Button appears green to indicate income
 - Tap on button
 - Lead to Tag Selection Page
 - Select tag(s)
 - Tap on Confirm Button
 - Back to Add Transaction Page with tags updated to reflect selected tags
 - Test Amount Field
 - Tap on Amount Field
 - Keyboard shows
 - Press Hide Keyboard Key
 - Hides keyboard
 - Entered message preserves in field

- Test Time Field
 - Default showing the current time
 - Tap on Time Field
 - Lead to Calendar Page
 - Select date and time
 - Tap on Confirm Button
 - Back to Add Transaction Page with time updated to reflect selected date and time
- Test Location Field
 - Default showing current location
 - Current location formatted as postal address
- Test Frequency Field
 - Default showing “One-time”
 - Tap on Frequency Field
 - Toggles among “One-time”, “daily”, “weekly”, “monthly” or “yearly”
- Test Repeat Number Field
 - If the current frequency is any of “daily”, “weekly”, “monthly” or “yearly”
 - Repeat Number Field shows
 - If the current frequency is “One-time”
 - Repeat Number Field does not show
 - Tap on Text Field in Repeat Number Field
 - Keyboard shows
 - Press Hide Keyboard Key
 - Hides keyboard
 - Entered message preserves in field
- Test Description Field
 - Placeholder showing “Description...”
 - Tap on Description Field
 - Keyboard shows
 - Press Hide Keyboard Key
 - Hides keyboard
 - Entered message preserves in field
- Test Camera Button
 - Tap on Camera Button
 - Camera Shows
- Test Confirm Button
 - Tap on Confirm Button
 - If amount is a decimal number
 - If amount is greater than 0
 - Lead to Main Page with number (and pig, if applicable) updated
 - If amount is smaller than or equal to 0
 - Alert Window showing warning of invalid amount
 - If amount is not a decimal number

- Alert Window showing warning of invalid amount
- Test Back Button
 - Tap on Back Button
 - Lead to Main Page with unchanged number and pig
- Test Transactions Page
 - Default showing the list of transactions in the current month
 - Displayed in reverse chronological order (i.e. Most recent on top)
 - Tap on any closed transaction cell in the list
 - Transaction cell unfolds
 - Shows type, amount, tags, date and time, location, description and picture
 - Tap on picture
 - Shows picture preview
 - Tap anywhere
 - Hides picture preview
 - Tap on edit button
 - Lead to Add Transaction Page (with edit mode)
 - Swipe left at the cell
 - Cell disappears
 - Tap on any open transaction cell in the list
 - Transaction cell folds
 - Shows type, amount, tags and date
 - Swipe left at a transaction cell
 - Delete Button appears
 - If the transaction is a one-time transaction
 - Tap on Delete Button
 - Cell disappears
 - If the transaction is a recurring transaction
 - Tap on Delete Button
 - Alert window shows
 - Tap on Delete Single
 - Cell disappears
 - Tap on Delete All
 - All cells of this recurring series disappear
 - Test Previous Button
 - Shows the list of transactions in the previous month
 - Top bar updates to show the month and year
 - Test Next Button
 - Shows the list of transactions in the next month
 - Top bar updates to show the month and year
 - Test Back Button
 - Tap on Back Button
 - Lead to Main Page
 - Scrollable
 - Test Trend Analysis Page

- Default showing empty chart with message “Choose time period to generate monthly trend analysis!”
- Tap on Select Date Buttons
 - Lead to Calendar Page
 - Select a date
 - Tap on Confirm Button
 - Back to Trend Analysis Page with line chart updated to illustrate monthly net income for the selected period updated to reflect selected date and time
- Tap on Tags Button
 - Lead to Tag Analysis Page
- Tap on Locations Button
 - Lead to Location Analysis Time Selection Page
- Test Tag Analysis Page
 - Default showing empty chart with message “Choose tags and time period to generate breakdown analysis!”
 - Tap on Choose Tags Button
 - Lead to Tag Selection Page
 - Select tag(s)
 - Tap on Confirm Button
 - Back to Tag Analysis Page with bar chart updated to illustrate expenditure breakdown by selected tags
 - Tap on Select Date Buttons
 - Lead to Calendar Page
 - Select a date
 - Tap on Confirm Button
 - Back to Tag Analysis Page with bar chart updated to illustrate expenditure breakdown by tag in the selected period
- Test Location Analysis Time Selection Page
 - Tap on Select Date Buttons
 - Lead to Calendar Page
 - Select a date
 - Tap on Confirm Button
 - Back to Location Analysis Time Selection Page
 - Tap on Confirm Button
 - Lead to Location Analysis Page
- Test Location Analysis Page
 - Shows a heatmap illustrating expenditure breakdown by location in the selected period
- Test Tags Page
 - Shows a nested table of all stored tags and their sub-tags
 - Tags sorted in alphabetical order
 - Tap on + Button at the top bar

- Prompt window shows
 - If the user input is empty or containing purely spaces
 - Alert Window showing warning of invalid amount
 - If the user input is not empty
 - New tag appears at the top level
 - Tap on + Button at a tag
 - Prompt window shows
 - If the user input is empty or containing purely spaces
 - Alert Window showing warning of invalid amount
 - If the user input is not empty
 - New sub-tag appears under the parent tag
 - Tap on a tag
 - Promot window shows
 - If the user input is empty or containing purely spaces
 - Alert Window showing warning of invalid amount
 - If the user input is not empty
 - Tag renamed to the given user input
 - Swipe left at a top-level tag
 - Delete Button appears
 - Tap on Delete Button
 - Cell disappears
 - All sub-cells disappear
 - Swipe left at a sub-tag
 - Delete Button appears
 - Tap on Delete Button
 - Cell disappears
- Test Set Budget Page
 - Tap on Text Field
 - Keyboard shows
 - Press Hide Keyboard Key
 - Hides keyboard
 - Entered message preserves in field
 - Tap on Confirm Button
 - If amount is a decimal number
 - If amount is greater than 0
 - Lead to Main Page with number (and pig, if applicable) updated
 - If amount is smaller than or equal to 0
 - Alert Window showing warning of invalid amount
 - If amount is not a decimal number
 - Alert Window showing warning of invalid amount

Detailed schedule + task list

Date	Milestone	Description	Person in charge
24 March 2019	Progress milestone 1	<ol style="list-style-type: none"> 1. UI Prototyping 2. Basic application flow between ViewControllers 3. Write up of Preliminary Design Document 	Lizhi
		<ol style="list-style-type: none"> 1. Set up project repository 2. Set up Travis CI and swiftlint 3. Implement "Transaction" model 4. Write up of Preliminary Design Document 	Fabian Terh
		<ol style="list-style-type: none"> 1. Set up PodFile 2. Implement StorageManager to support basic saving and loading. 3. Write up of Preliminary Design Document 	Travis Ching
7 April 2019	Progress milestone 2	<ol style="list-style-type: none"> 1. Implement transaction recording, editing and deletion (Front-end) 2. Implement transaction history (Front-end) 3. Implement Budget creating, budget status and mascot Bacon (Front-end). 4. Implement transaction prediction mechanism (Front-end + model + logic) 	Lizhi
		<ol style="list-style-type: none"> 1. Complete Transaction model 2. Implement Observer pattern 3. Implement smart location driven prompt feature 	Fabian Terh
		<ol style="list-style-type: none"> 1. Complete Transaction storage and logic (Back-end) 	Travis Ching

		<ol style="list-style-type: none"> 2. Implement Budget (Back-end) 3. Implement Location and Image saving (Back-end) 4. Implement CoreLogic 	
17 April 2019	Progress milestone 3	<ol style="list-style-type: none"> 1. Implement expenditure trend analysis over time. (Front-end + logic) 2. Implement expenditure breakdown by tag. (Front-end + logic) 3. Implement expenditure breakdown by location. (logic) 4. Implement Recurring Transaction (Front-end) 5. Implement Tag adding, editing and deletion (Front-end) 	Lizhi
		<ol style="list-style-type: none"> 1. Implement TagManager and Tag model 2. Implement HeatMap (expenditure breakdown by location front-end) 3. Rigorous testing 	Fabian Terh
		<ol style="list-style-type: none"> 1. Implement Tag-Transaction database schema 2. Implement Recurring Transaction (back-end) 3. Implement Prediction (back-end) 4. Complete CoreLogic 5. Rigorous testing 	Travis Ching
21 April 2019	Final Project Report	<ol style="list-style-type: none"> 1. UI/UX touch-up 2. Code refactoring 3. Code Documentation 4. Final Project Report write up 	Lizhi, Fabian Terh, Travis Ching