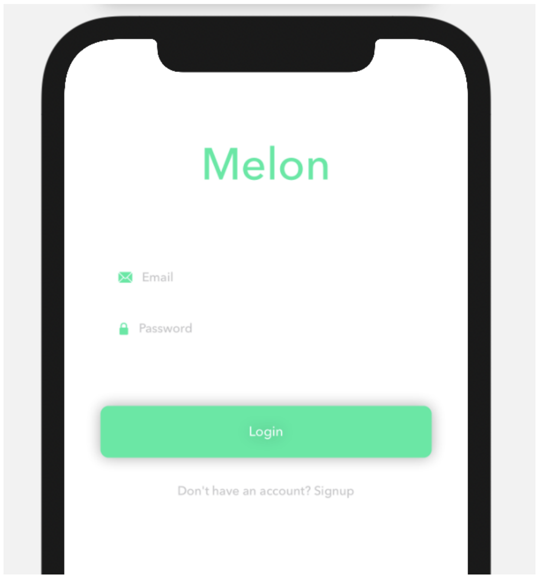
MELON

Authors
Mihir Wadekar, Shreya Gupta, Chaitenya Gupta, Josh Arnold

Introduction

The goal of our project is to develop an application centered around taxes & students. Students face challenges when dealing with taxes—for example, not knowing what forms to file, being pay-walled by different services, and having their questions answered. Our goal is to develop a solution that accommodates student needs entering the tax world for the first time.



Above is an example of our beta version product. We entitle our solution, **Melon**.

Our app adheres to a few core principles: 1) we want great design, and great user experience. We believe the experience for filing taxes can be greatly improved if a smooth, efficient user experience is developed. The story behind our name is simple: watermelons are beautiful fruits: green on the outside, red on the inside. Just like a watermelon, we strive to make our app beautiful (and green). A beautiful, fun app, makes the tax filing process more engaging.

Furthermore, we abstract the unnecessary actions users typically take when filing a tax form, such as trying to understand difficult questions, double checking if everything is filled out, etc—we provide simple questions and handle form validation to make a smooth process.

In addition, we decided to make our project open source. This is so that people across the world can contribute to our effort to build a quality tax platform that is student-friendly. Also, as individuals in our team, we can further develop our skills by working on a large scale open source project, including writing quality documentation, writing robust, concise code, and dealing with git (merge requests, push-pull, rebases, etc!).

Before tackling the technology survey, we would like to discuss some legal limitations that we have faced that have shaped the direction of the final product.

Legal Limitations

Our product allows free access to a better and flushed out user experience for filing out tax forms, allowing people with lower income to easily fill out their taxes without being paywalled. However, it does entail different legal concerns. Firstly, in our database, we store **user-sensitive information**. For example, everything from name, to address, country of birth, social security, etc. Although users have to be authenticated to access our server, our app is not yet secure from cyber-attacks and it is possible for hackers to find a way into our system to read this sensitive information. Users may also be concerned about developers having access to their information.

To address this primary security/legal issue in our timeline, we have decided to separate out parts of the functionality of the app into a separate application. The second application (hereby referred to as Nebula) is a secure communication application that allows users to have private conversations that are deleted after a set time limit. This was initially a feature meant to be part of the original application. We made the decision to separate out the two based on 5-6 failed attempts of submitting the application to the Apple app store due to legal complications. As all of our developers are international students, this issue was not something we were able to tackle in the current timeframe. Since the application is open source, we hope that other developers are able to find a solution to this problem in the future.

In the future, one direction our team would like to explore is testing the robustness of security: we would like to try various, common security attacks and **benchmark** how robust our product's security is. The bug bounty program will help us in this regard by providing ethical hackers an incentive to try and break our system, and alert us of any security vulnerability they might encounter by providing a pull request to the repo. Prompt action will be taken by the team then to patch this vulnerability.

Technology Survey

In this section, we briefly entertain a discussion and reasoning behind our tech stack. We divide our discussion into two sections: frontend vs. backend. Our product doesn't have too much in terms of middle-ware.

Frontend

The first major discussion point for a mobile app is whether or not to build a cross-platform product vs. a native product. Let's just go ahead and end this discussion right here. Cross platform development, to put it politely, is disgusting. Firstly, it is in-efficient. When you scroll any type of view on a cross platform mobile app, often, it is below the gold standard of 60 fps. Secondly, it is difficult to take advantage of native features. For example, Swift and the iOS platform has a bunch of accessibility features including haptic touch, dark mode, resizable fonts, etc—these cross platform frameworks often don't have all of these features baked into their APIs, and if not, you have to regress to sketchy, non-maintained third party libraries to implement such functionality.

The benefit of cross platform development is you get to deploy to two platforms. However, our team values the design principle of minimal interface design: specifically, we try to push as much business logic to the backend as possible: thus, the front-end is really just managing a user-interface and its state. Building an android app, for example, should be fairly straightforward as most of the business logic would have already been written.

Lastly, we must discuss what platform we wish to build for. Apple, iOS, Swift, and SwiftUI - undeniably these frameworks & platforms provide the most beautiful experience to the user on the market. Our team values creating a beautiful experience, thus, it is clear we must write our product in Swift!

Now, when dealing with Swift, there is a new heated debate as to whether to use SwiftUI or UIKit. Let's again address this candidly: SwiftUI is INSANE. It is completely out of the wazoo, powerful. UIKit users that hesitate to jump to SwiftUI are resistant to change. Yes, SwiftUI has its limitations, however, from our experience, you can achieve 90% of what you can do in UIKit. The only case you shouldn't be using SwiftUI, is if you are working at a company that has millions of users and performs mission critical tasks, where stability is a MUST. However, any agile, indie team should 100% adopt SwiftUI. This isn't a discussion. It is a fact.

Based on this reasoning, we decided to use SwiftUI for our main application (Melon). For our second application, we decided to try out UIKit to overcome some of the limitations of the former.

Backend

In terms of our backend design, we had multiple ways to approach our stack. Again, we chose the most proactive approach, that is easy, simple and robust.

In terms of user-authentication and general data-persistence, there are a myriad of solutions available for developers to implement, for example, Firebase by Google. For Melon, we chose a reliable proven stack using Parse Server, a Node.js w/ Express server wrapped around a MongoDB backend. MongoDB is a fantastic mobile backend. Parse server makes it simple to get up and running.

```
        let user = PFUser()
        user.username = username
        user.email = username
        user.password = password
        user["name"] = firstLast

        user.signUpInBackground { (success, error) in
            // ...
        }
```

The above example is native Swift code. Through Parse's open source backend API, look how easy it is to create and authenticate a new user. Custom properties can be adhoc set on the user just like using a dictionary. For our second application, Nebula, we chose to continue with Firebase by Google since it provides real time communication, which Parse does not offer.

We decided to host our Parse/Node.js server on a third party website, back4app, which provides free hosting for small projects!

Another consideration we had to have was how we should handle niche, business logic tasks, such as converting a dictionary into a baked PDF file. Yes, we could do this natively in the iOS app, but it makes more sense to get the front-end as light-weight as possible. For a server that requires a few basic routes, HTTPS requests and the like, it makes sense to use a Flask Server. Flask is written in Python and takes literal seconds to get up and running. Plus, working in a Python environment is really fast in terms of development time.

The below code snippet shows how minimal part of our flask server is!

```
app = Flask(__name__)

@app.route('/user/<user_id>/form/<form_id>', methods = ['GET', 'POST'])
def form(user_id, form_id):
    fillPdf = fillPDFData()
    fillPdf.exec(request.json, "8843")
    email = emailSend("8843", request.json["email"])
    email.exec()
    return {"test" : "key", "dict" : "d"}
```

Overall, we chose technologies we were mostly familiar with, to speed up our development time.

We noticed one solution, Realm (also by MongoDB), seems to be one of the gold standard modern mobile databases, however, its complexity is significantly more than our current proposed solutions. If given enough time, it would be fantastic to learn about Realm and how to use it, but for now, we will stick with our current approach.

Project Management Software

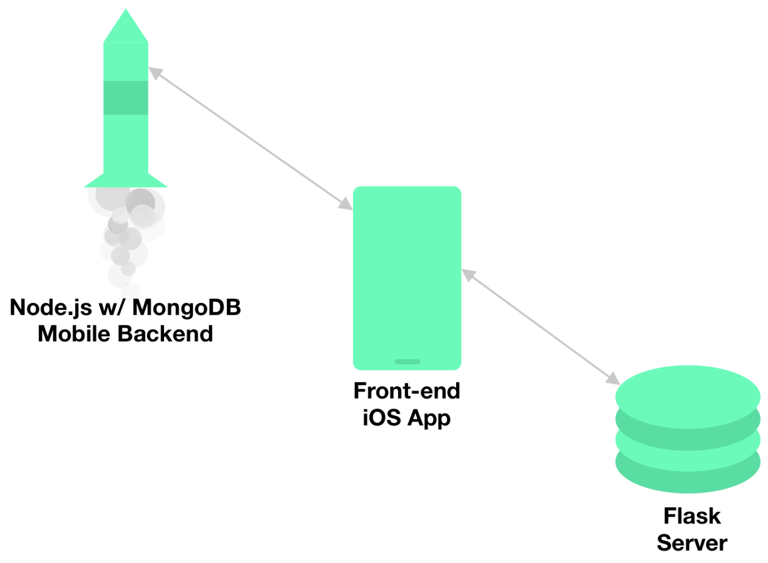
Since our team is only four people, we can communicate fairly efficiently through a group chat (platform: Facebook Messenger). This is nothing fancy, but it honestly gets the job done. GitHub is a must for managing code contributions between us. We also can make use of GitHub issues in terms of maintaining a todolist. Again, nothing fancy here, but these tools are fairly obvious to be great solutions for our team!

System Architecture

We made careful design choices when architecting our product. There are two levels to our architecture:

1. Macro architecture
2. Micro architecture

The macro architecture is concerned with the high-level design between front-end and back-end, including how the data flows between the two. The micro architecture is concerned with individual design choices for each component, for example, how the iOS app is designed. We will first discuss the macro architecture.

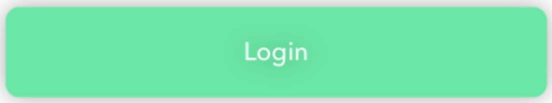


The figure above highlights our macro-architecture. There is bi-directional data flow between the iOS app and both backends. The Flask Server is a lightweight backend server that processes logic specifically related to filing taxes. The Node.js w/ MongoDB backend is the core backend, that handles user authentication and object persistence. We use the open source software, entitled Parse Server*,* which in actuality, is a pre-written wrapper for MongoDB that is available as a native framework in swift.

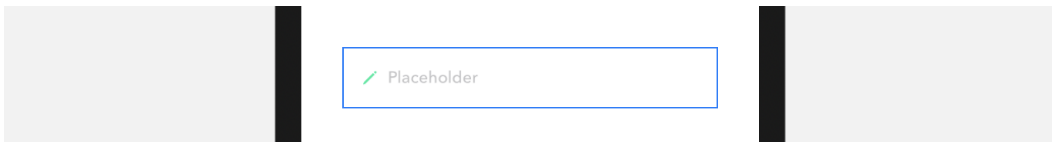
The MongoDB backend is amazing in our situation, specifically for developing a light-weight, scalable, flexible application. Specifically, you do not need to pre-define any database schemas, as MongoDB is a NoSQL relational database. Thus, it is more forgiving, and has sped up our team's development cycles significantly. Ultimately, our macro architecture is quite simplistic.

Our iOS app architecture is quite unique & modern. Typical mobile architectures follow either a **MVC** (Modal-View-Controller), **MVVM** (Model-View-ViewModel), or even a **VIPER** (View-Interactor-Presenter-Entity-Routing) architecture. However, as we are using **SwiftUI**, a **declarative** programming language, we follow a similar architecture adopted within **React.js** apps which is to use a **redux-esque architecture.**

Firstly, our app has different views. One view may be the login screen, another view may be the user's profile, for example. In order to construct these views, we first develop a library of components. Any component that will be reused we consider a component. For example, a button (as shown below), will be developed as a reusable view.

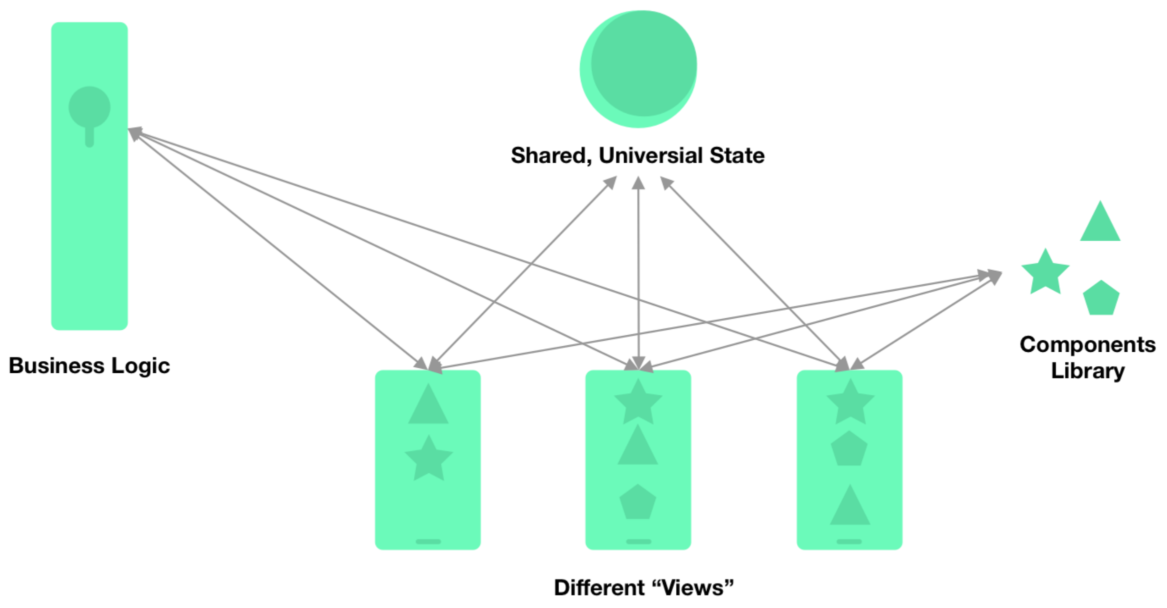


The code for each component is relatively concise. Perhaps, ~30 lines of code. Components can also reuse other components too. By curating a library of UI components, we keep our code-base relatively flat and non-hierarchical—components scale horizontally, instead of nesting deep dependencies when developing views. The trade off for this, of course, is we have a large number of independent swift files in our project. When we develop different Views, such as a login screen, it is as simple as placing a view component in a vertical stack. Thus, our view code is relatively concise too.



Above is another example of a textfield component. The blue box around it is not shown during runtime in the app. The placeholder and image to the left are easily customizable.

There was the slight challenge of defining a criteria for what "counts" as a view, vs. a component. There are some cases in which we have a UI element that is too small to be a view, but a little too big or a little too niche/specific to be a component. In this case, we just treat this UI element as a component in the case it <u>might</u> be reused in the future.



The figure above shows the micro architecture for our iOS app. You can see the components library and how the components can be rearranged in different ways to form different views. Minimal business logic exists within each view, to keep the code from getting too bloated. The beautiful aspect of our app is the Shared, Universal State. Typically, apps have spaghetti dependencies between different views, especially in terms of navigation or displaying an overlay, such as a UIAlertView. Each view, however, is passed a universal state object. If a view wants to navigate to a different view, its as simple as updating the universal state like so:

```
appState.currentScreen = AppView.home
```

We shift the navigation logic solely to the App State, so that if we need to change/update the navigation logic, we can do so internally without needing to update the public-facing API. Furthermore, views can now navigate to any page in any order. We do not have to worry about memory issues as there is only one view kept in memory at a time, when a new view appears, the other view is removed (unlike the MVC architecture, for example).

The shared state is also useful for abstracting other common behaviour, such as presenting an alert to the user:

```
appState.alertMessage = error?.localizedDescription ?? "Something
went wrong!"
```

By setting the `alertMessage` property on the app state, the app state will automatically handle displaying an alert to the user. Previous applications would have to instantiate a view, and handle adding it and removing it from the hierarchy, however, our architecture abstracts all of that!

Finally, there is the business logic shown in the previous diagram. Business logic includes calling one of our databases, or calling a large-complex function. We simply abstract these logic into different classes (typically as a set of static functions on a class) that any View can call.

This, again, is to remove bloating.

Ultimately, our architecture is clean, scalable, and loosely coupled, adhering to quality, modern software engineering design.

One criticism about our iOS app microarchitecture is the shared state is perhaps not thread-safe, and exposes state unnecessarily. Many programmers are against global state, however, given our use-case, we think global state can make a lot of sense, specifically because we are not dealing with multiple threads, only simple navigation and common behaviour patterns, in which global state makes development extremely clean and fast!

User Stories

Priority	User Stories
HIGH	As a user, I want to be able to sign up (create an account) if I don't have an account, and login and out of the app. Logging out of the app additionally ensures that my data will be protected when I am not using the app.
HIGH	As a user, I should be able to submit a Tax form (1040 / 1040 NR) through a series of easy to use screens and simple questions.
HIGH	As a user, each piece of information should be broken down into small, simple questions that I can optionally skip and come back to.
HIGH	As a user my tax form should be easily accessible. I would like the form to be emailed to me, with all data filled in, after I complete filling out the questions.
HIGH	As a user, I should be able to trust that my personal information will be protected and inaccessible to anyone other than me.
HIGH	As a user, I should be well informed about upcoming dates and deadlines so I don't forget to file any specific forms on time.
HIGH	As a user, I should be able to communicate privately and securely with other users on the app (e.g. tax-prep related conversations for companies, non profits, etc.)
MEDIUM	As a user I should be able to edit my profile from the app.
MEDIUM	As a user the "state" / progress of my tax application should be persistent throughout multiple devices.
MEDIUM	As a user, I would like to fill in my data in a "smart" way, where the app provides me automatic suggestions of what I am entering, e.g, auto completing my address, autofilling phone-numbers & email, etc, etc.

LOW	As a user, I should be able to upload my W-2 form and have parts of my 1040 automatically filled out for me.
LOW	As a user, I should be able to E-file taxes from the app.
LOW	As a user, I should be able to ask simple questions and get quick responses regarding the tax forms.
LOW	As a user, I should be able to help respond to other people's inquiries if I have any information that I can share.
LOW	As a user, I should know whether or not I can trust the responses provided to me regarding my questions (or questions by the public).

Technologies Used

In this section, we highlight as many of our technologies as possible!

Backend

Flask

Parse Server

Languages

Swift

Python

Ruby (For cocoapods)

Javascript (Node.js backend)

Frontend Frameworks

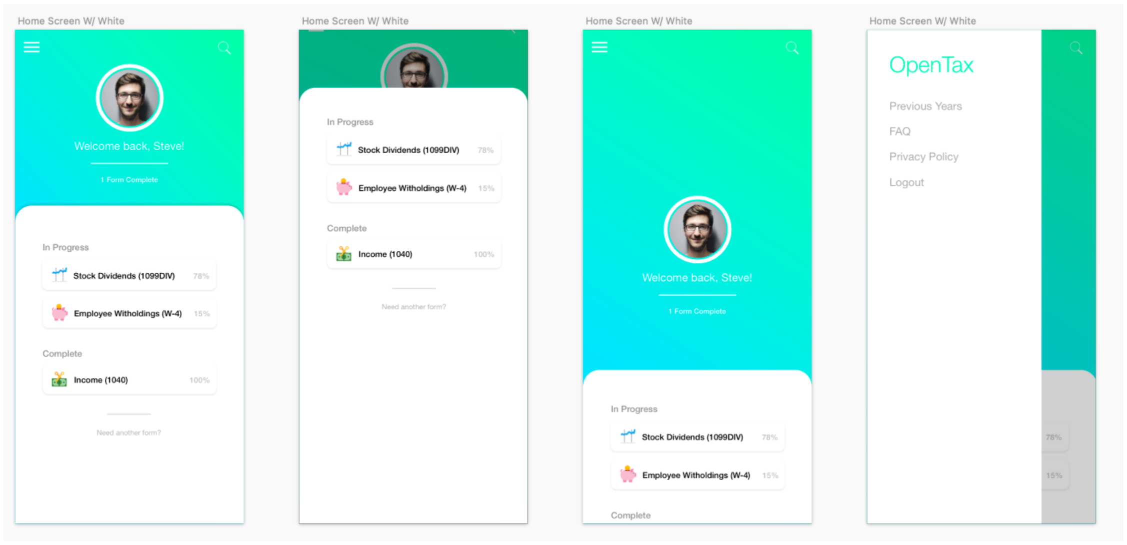
SwiftUI

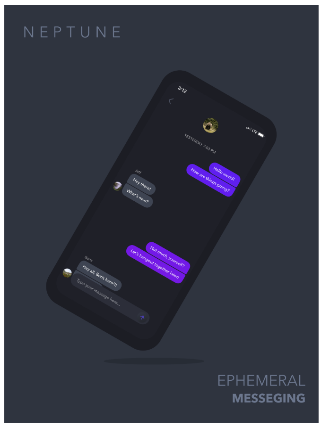
Cocoapods

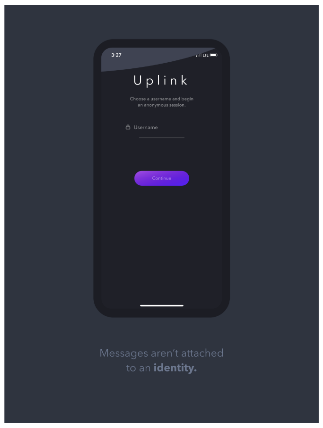
Alamofire

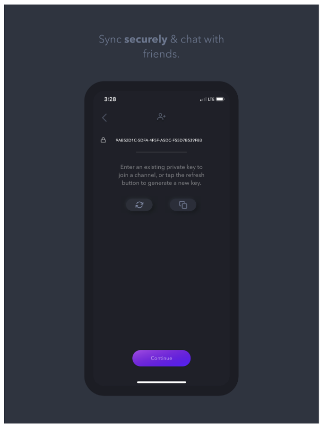
Lottie-iOS

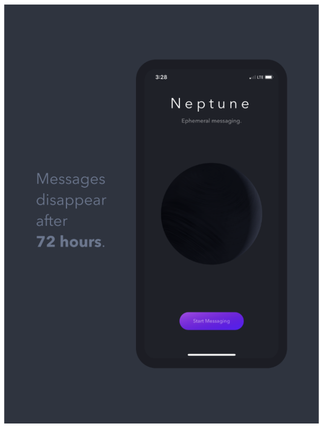
Quick Look

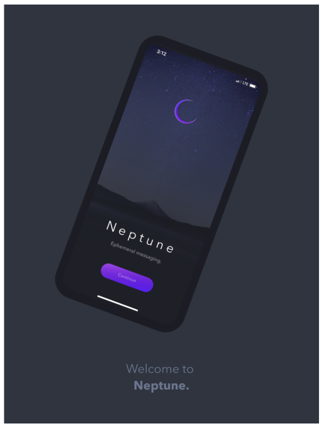

Home Screen W/ White
Welcome back, Steve!
1 Form Complete
In Progress
Stock Dividends (1099DIV) 78%
Employee Witholdings (W-4) 15%
Complete
Income (1040) 100%
Need another form?
OpenTax
Previous Years
FAQ
Privacy Policy
Logout


NEPTUNE
EPHEMERAL
MESSEGING


Uplink
Username
Continue
Messages aren't attached
to an identity.


Sync securely & chat with
friends.
Continue


Messages
disappear
after
72 hours.
Neptune
Ephemeral messaging.
Start Messaging


Neptune
Ephemeral messaging.
Continue
Welcome to
Neptune.

Cost Analysis

For the user, the cost is **minimal**. Since the app is free and open source, the main cost is the space that the program takes up on the user's phone.

Similar to the user, since the app is open source, developers have access to the entire source code for the app. This, combined with the public GitHub repository, makes collaboration amongst various developers easier.

The **primary cost** will come from the cost of running and maintaining the servers, and managing the app itself. Managing the app can include identifying and fixing bugs, conducting user feedback, implementing new features, and improving the design. In order to further improve the code for the application the team plans to provide small bug bounties for open source developers to spend their valuable time to fix old features and add new features to the app.

Here's a breakdown of the expected costing for the team

- Parse Server (Silver Plan for Production Apps) - $200 USD per month
- Bug Bounties and grants for open source devs - $1000 per month
- Heroku (Production Plan) - $7 USD per month.

Total expected cost per year = $14,484 USD

Since the project is open source, the team doesn't have a sustainable way to generate revenue on its own. As such we would go the non-profit route and accept tax-exempt donations to the project to further the development of a free tax app to help users file their taxes. This is also something that users can pay for through the app. **100% of the money will go towards funding the app's maintenance, development and bug bounty programs**.

Conclusion

This project is inspired by premium tax software like TurboTax, and the team is on a mission to remove corporate greed for something as basic as filing taxes. TurboTax users are bombarded with options to buy costlier packages, which they may not even need. Melon on the other hand just wants to be an application platform for the open source community to elevate and simplify the complex process of tax filing.

We want to provide younger generations with an easy to use and efficient method that removes the complexities of the modern tax system, saves time, and improves the quality of experience while working on tax applications. Since virtual communication is a vital part of our users' daily lives, we also want to provide a platform that encourages conversations in the same place, with a guarantee of privacy and security that other large applications such as Messenger and Whatsapp are not able to provide.

Thank you for reading our requirements document. Please find our GitHub link below, here: https://github.com/Bearers-of-Worldly-Spirits/main_app