Valentin Silvera // Capstone Project

Ranked App - Technical Documents

Ranked is a native iOS app that allows users to quickly create and share polls with people around them. The name comes from the voting system it uses: ranked voting. With this, users can order any amount of options by preference, or leave them unranked. After everyone has voted, the poll can be closed (currently manually, automatically in the future), and the votes are counted. The specific method used is instant-runoff voting. This emulates the effect of separate ballots per round:

1- On the first round the votes are counted as they come

2- If there's a majority (50% + 1), that's the winner

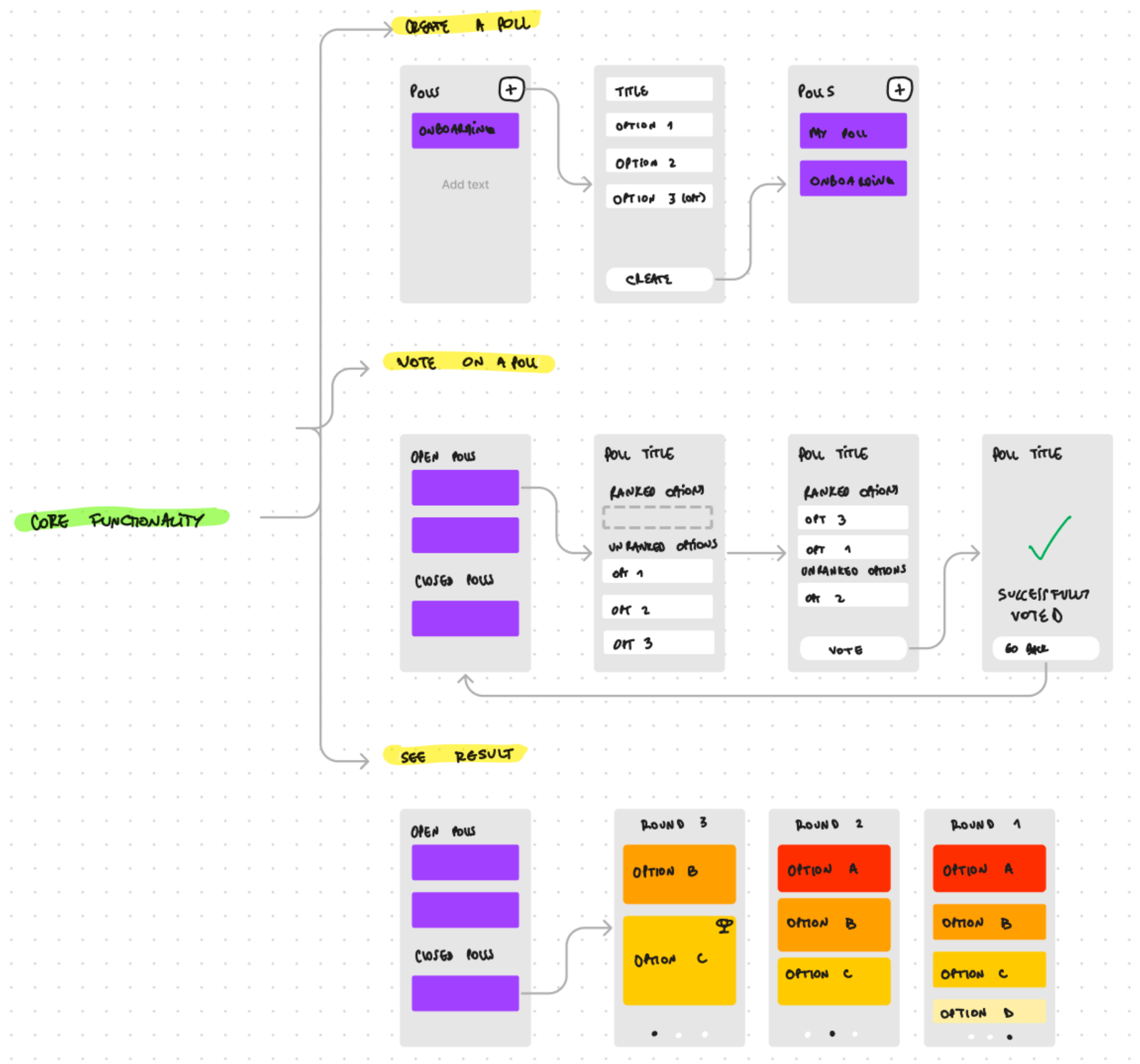
3- Otherwise, the option with the least amount of votes gets eliminated from every ballot (or array)

4- Repeat from step 2 until a majority is reached or there's a tie

Architectural Decisions

Before starting the design and development of the app, there were constraints put in place to define the application:

- Programming Language: The project needed to be built fast, so I chose to use **Swift**. Swift is an open source programming language that allows a fast development process. It's expressive, so it requires less code than Objective-C. Automatic Reference Counting (ARC) makes memory management a non-issue (as long as we are careful with memory leaks, like retain cycles). Furthermore, Swift is a safe language, by being strongly-typed, it handles most errors in compile time, instead of run time. Shortening the feedback loop and preventing many bugs to make it to production. The main downside of using Swift is that it's not cross-platform.

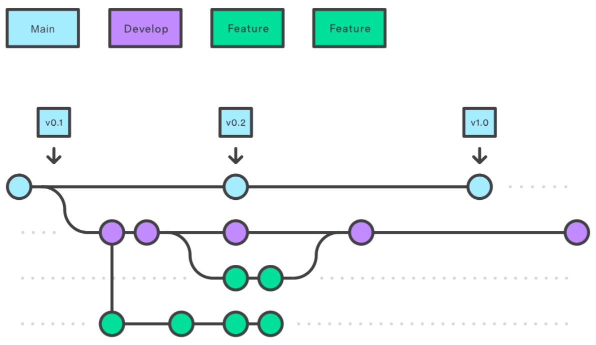


Img 1. LoFi prototype with the UX flow of the app

If this project was to turn into a business, the next step would be to develop an Android version of the app with Java, Kotlin, or similar.

- UI Framework: Here the two main options were using UIKit or SwiftUI. I chose **SwiftUI** because it offers the most consistent user experience, it looks great, and it's incredibly concise to write thanks to its declarative syntax. The downside is that there are many elements (eg: lists) that are not very customizable. But thanks to it's interoperability with UIKit, you can still create custom views. Furthermore, the new lifecycle management has been greatly simplified. As now we can add modifiers to view such as .onAppear(perform:) to manage app states.

- Version Control: In any software development more complex than a "hello world" there should be some version control in place. I chose to use a custom **GitFlow** as my workflow: the main branch is where the most stable code is, and there's no develop branch to avoid useless complexity; features and fixes have their own branches with the appropriate "fix/ or feat/" prefix, to prevent braking changes in the main one.



Img 2. Gitflow Workflow. From Atlassian

Data storage

For the data storage there are two main parts: on-device and cloud storage.

On-device storage for Swift can be done in different ways, the main first-party ones being CoreData and User Defaults. An example of User Defaults in the app is the username when creating a poll:

```
@AppStorage("CREATOR") private var creator = “” // creating and storing
the variable

TextField("Write your name as a creator of the poll...",
          text: $creator) // using the stored variable
```

User Defaults should be used for small data (eg: Strings or Booleans) because they get loaded on app launch, but using them is very simple. For more complex data, such as the onboarding poll, CoreData is a better solution.

Cloud storage is a different topic. First and foremost I had to choose between SQL and NoSQL. I went with **NoSQL**, again for quick development purposes. I didn't need to specify a schema at the beginning, providing much more flexibility for the many changes I made to the model during development (ie adding and removing fields). Other benefits of using NoSQL is easier to scale, decentralized, and redundant, query speed (specially for non-complex queries), among other benefits. Some drawbacks could be the lack of a standardized language and a smaller user community (for now).

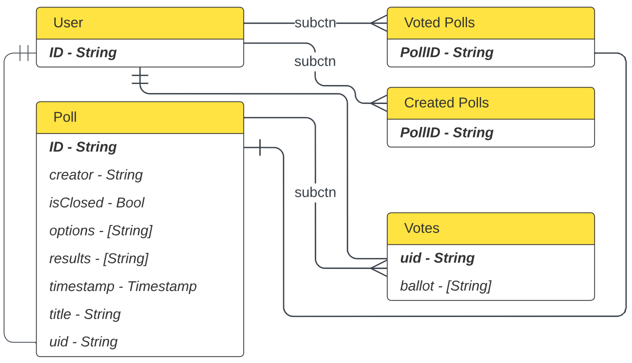
With so many services it can be more difficult to make a decision. I chose Firebase **Firestore**, a NoSQL document database that is fast and serverless. At the current scale it's free to use, easy to implement, and very robust. Implementing it in the code is a simple process:

- Import the dependencies, this can be done with Cocoapods, Cartage, or Swift Package Manager, which is the one I used.

- Create the project using Firestore's console

- Setting up the app with the database (registering the app and importing the plist)

For the user authentication I'm using Firebase's anonymous authentication. This provides a smooth and secure way for users to retrieve the polls they created and voted on, without the need to provide an email or using social accounts. But most importantly, it allows users to start using the app right away, instead of being met with a wall on start up that asks them to provide credentials, before even being able to test the product itself.. It delights and provides the "it just works" aspect of the app. Of course this doesn't allow for account migrations, multi-device accounts, or continuity between platforms. But Firebase offers a very simple anonymous-to-permanent-account conversion (not yet implemented). On the database side, the user collection stores the user id (uid) and two subcollections: created and voted on polls, both storing the poll id that has been created or voted on, respectively. The poll collection stores the poll data (id, creator, options, etc), as well as a sub-collection storing the votes for that poll, including the uid of the user that submitted that vote.



Img 3. Database Schema

Security

Security can be an extremely wide topic and I could make an entire document just talking about it for this project. Because most of the sensitive data including authentication is done with Firebase, I will be focusing there.

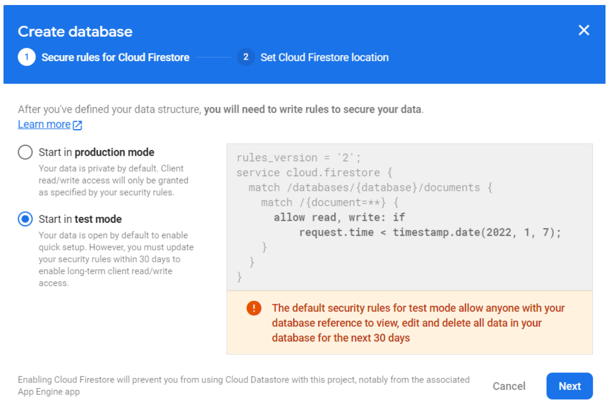
Firebase Auth: this is the authentication system provided by Firebase, and the one I'm using in the app. This service has been audited before, and the main takeaways on security risks are:

- It lets users choose weak passwords by default: the only requirement is that it has to be longer than 6 characters. So a user could potentially write "123456" and it would be allowed. The entropy of this password is too low and doesn't comply with OWASP rule MSTG-V4.4. Requiring more complexity on the UI still doesn't solve the backend issue, and would fail an audit. The solution for this would be to stick to using a social media log-in (eg: Apple, Google, Facebook, etc)

- OWASP rule MSTG-V4.5 dictates that the backend implements an exponential back-off after an excessive submission of log-in attempts. Firebase does this but is not clear about what limit this is, which might also fail a security audit.

- 2FA is not easy to implement, which opposes OWASP rule MSTG-V4.8. The possibility to use it is there but it needs Google Cloud's Identity Platform, while it should be out-of-the-box. Meanwhile, if a third party auth provider (Google, Apple, etc) uses multi-factor authentication, one can't say the app uses it, as the user can disable it on those platforms.

After carefully weighing the use cases and possible sensitivity of the information on the app, I still decided to go with Firebase Auth as I consider it to be secure enough for the intended use of the app.


Create database
1 Secure rules for Cloud Firestore
2 Set Cloud Firestore location

After you've defined your data structure, **you will need to write rules to secure your data.**
Learn more

[ ] Start in **production mode**
Your data is private by default. Client read/write access will only be granted as specified by your security rules.

[✓] Start in **test mode**
Your data is open by default to enable quick setup. However, you must update your security rules within 30 days to enable long-term client read/write access.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2022, 1, 7);
    }
  }
}
```

The default security rules for test mode allow anyone with your database reference to view, edit and delete all data in your database for the next 30 days

Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project, notably from the associated App Engine app

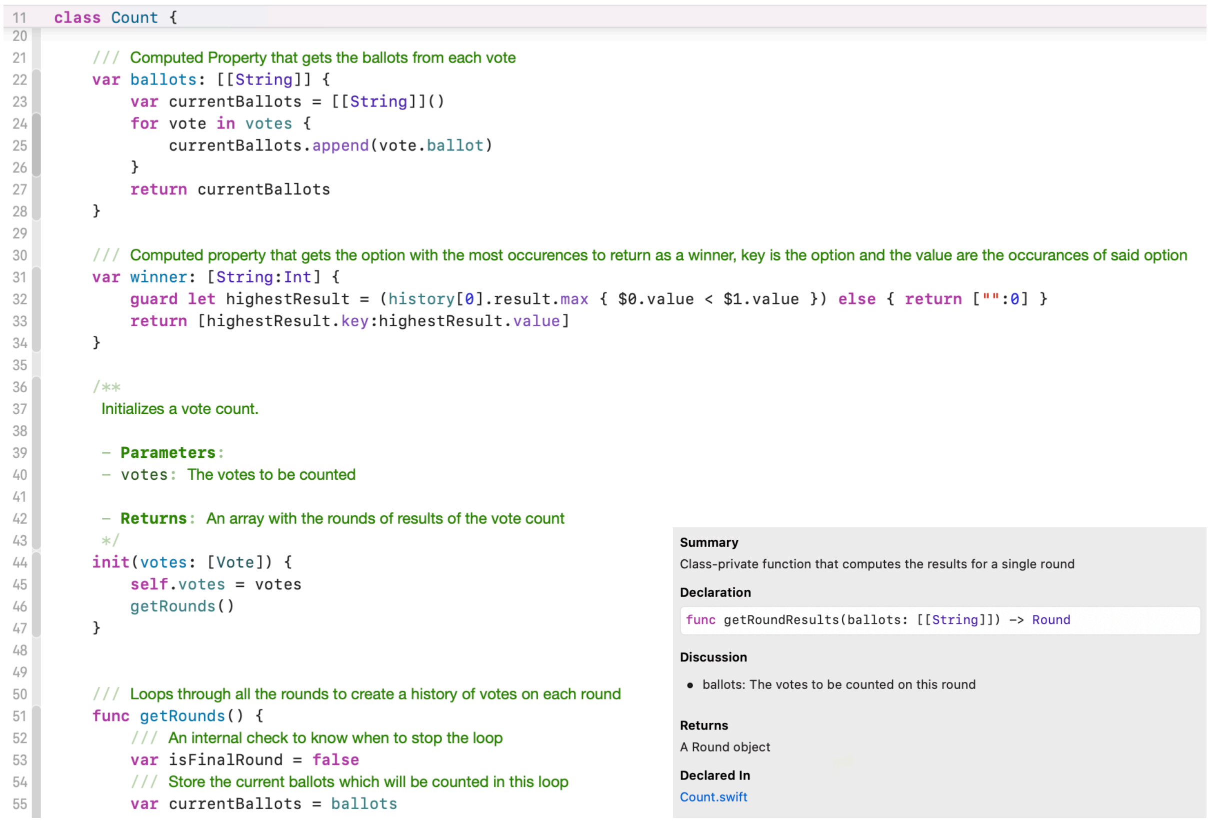
Img 4. Firestore's security mode dialog

Firebase Firestore: on it's own, Firestore is extremely secure (as it should be expected from Google). But there are development decisions that can make it otherwise. And the platform encourages those poor decisions. I'll explain why:

When creating a new project, Firebase presents us with two options: either starting in test mode, which sets write and read rules to allow anyone with access to the database to add, modify or delete anything; or production mode, which doesn't allow anyone to do it. This encourages developers to architect their data without really thinking about security, which can (and does) make it more difficult to do so later. Furthermore, Firestore doesn't have a typical API, instead, the app acts directly on the database with single-use API methods (like uploadPoll() or fetchPolls()). For this, security rules can be very complex to write, as I am myself struggling with these. For example, it's easy to allow a single user (ie the person who created the poll) to modify it, but more difficult to allow a group of users (ie people who have been shared a code to vote on a poll) to modify it. Firebase is targeted to beginners, and while it's easy to use and implement, it's not so straightforward to get it right. Having said that, there's no service I'd rather use when I start a new project and need to submit results quickly (as in the case with this project).

The algorithm

The algorithm is very well documented in Swift already, as it allows Xcode to generate documents automatically:



```swift
class Count {

    /// Computed Property that gets the ballots from each vote
    var ballots: [[String]] {
        var currentBallots = [[String]]()
        for vote in votes {
            currentBallots.append(vote.ballot)
        }
        return currentBallots
    }

    /// Computed property that gets the option with the most occurences to return as a winner, key is the option and the value are the occurances of said option
    var winner: [String:Int] {
        guard let highestResult = (history[0].result.max { $0.value < $1.value }) else { return ["":0] }
        return [highestResult.key:highestResult.value]
    }

    /**
     Initializes a vote count.

     - Parameters:
     - votes: The votes to be counted

     - Returns: An array with the rounds of results of the vote count
     */
    init(votes: [Vote]) {
        self.votes = votes
        getRounds()
    }


    /// Loops through all the rounds to create a history of votes on each round
    func getRounds() {
        /// An internal check to know when to stop the loop
        var isFinalRound = false
        /// Store the current ballots which will be counted in this loop
        var currentBallots = ballots
```

Summary
Class-private function that computes the results for a single round

Declaration
```swift
func getRoundResults(ballots: [[String]]) -> Round
```

Discussion
- ballots: The votes to be counted on this round

Returns
A Round object

Declared In
Count.swift

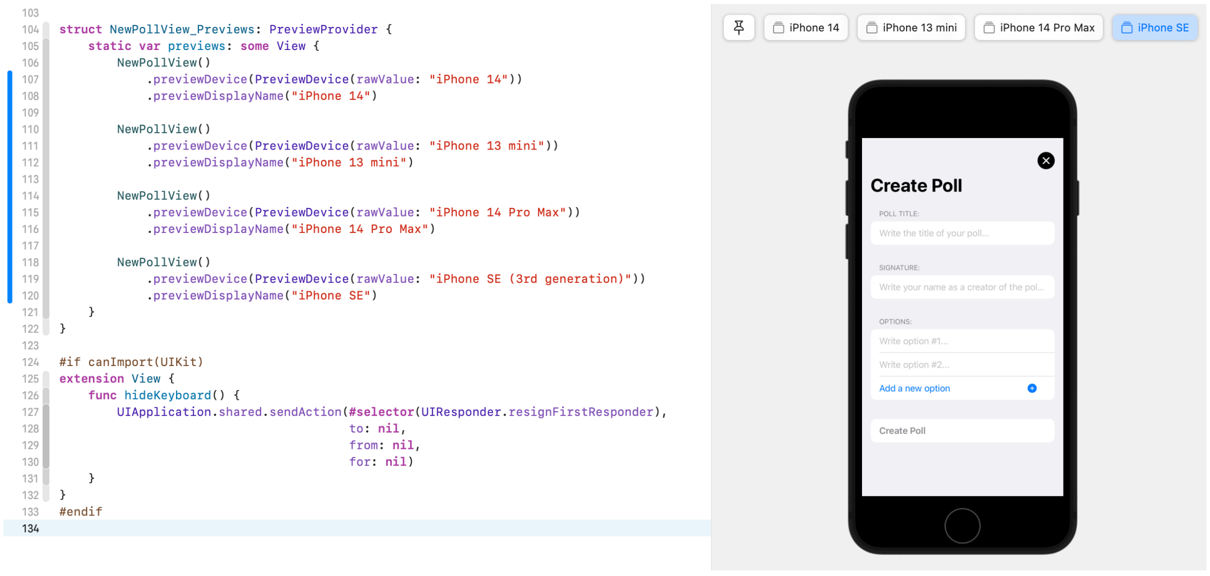
Img 5. Inline documentation of the algorithm in Swift using Swift-specific markdown. On the lower right a sample of documentation generated for the getRoundResults() function by Swift with the inline docs

Testing

Testing is a fundamental part of development, so important that there's an approach called Test Driven Development. TDD is a great tool for projects that saves on work on the long run, while taking some extra time at the beginning of the project (which is why I didn't use this approach for this project).

Testing can be divided in two big categories: manual (device and simulator) and automated testing.

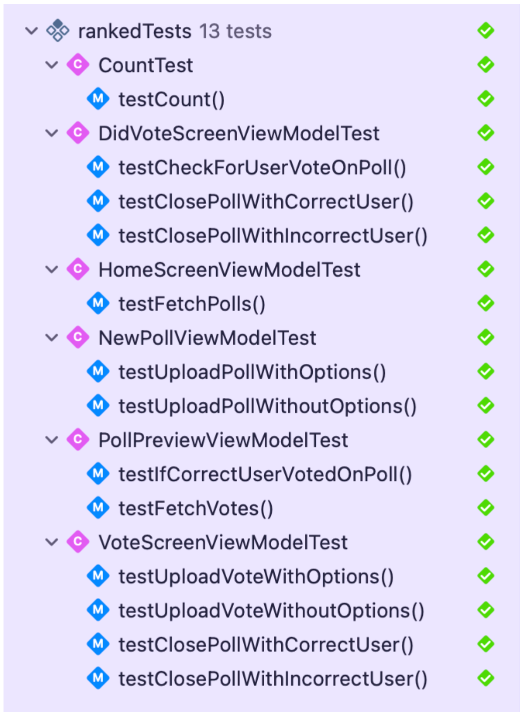
Manual testing (ad hoc):



```swift

struct NewPollView_Previews: PreviewProvider {
    static var previews: some View {
        NewPollView()
            .previewDevice(PreviewDevice(rawValue: "iPhone 14"))
            .previewDisplayName("iPhone 14")

        NewPollView()
            .previewDevice(PreviewDevice(rawValue: "iPhone 13 mini"))
            .previewDisplayName("iPhone 13 mini")

        NewPollView()
            .previewDevice(PreviewDevice(rawValue: "iPhone 14 Pro Max"))
            .previewDisplayName("iPhone 14 Pro Max")

        NewPollView()
            .previewDevice(PreviewDevice(rawValue: "iPhone SE (3rd generation)"))
            .previewDisplayName("iPhone SE")
    }
}

#if canImport(UIKit)
extension View {
    func hideKeyboard() {
        UIApplication.shared.sendAction(#selector(UIResponder.resignFirstResponder),
                                        to: nil,
                                        from: nil,
                                        for: nil)
    }
}
#endif

```

Img 6. SwiftUI's PreviewProvider allows us to render the view we're working on in real time, with different devices and configurations

Manual testing consists in running the app in a simulator or real device, and testing the entire flow, or parts of it. It's important that not only the developer does this, but also users, as use cases can vary. This can be aided with the help of TestFlight, which allows users to download beta versions of the app, and submit quick reports by just taking a screenshot, as well as automatic crash reports. This has not been implemented yet due to time constraints. Testing the UI in real time SwiftUI makes it easy to view changes in many different devices while building it, thanks to PreviewProvider and the Canvas view.



- rankedTests 13 tests ✓
  - CountTest ✓
    - testCount() ✓
  - DidVoteScreenViewModelTest ✓
    - testCheckForUserVoteOnPoll() ✓
    - testClosePollWithCorrectUser() ✓
    - testClosePollWithIncorrectUser() ✓
  - HomeScreenViewModelTest ✓
    - testFetchPolls() ✓
  - NewPollViewModelTest ✓
    - testUploadPollWithOptions() ✓
    - testUploadPollWithoutOptions() ✓
  - PollPreviewViewModelTest ✓
    - testIfCorrectUserVotedOnPoll() ✓
    - testFetchVotes() ✓
  - VoteScreenViewModelTest ✓
    - testUploadVoteWithOptions() ✓
    - testUploadVoteWithoutOptions() ✓
    - testClosePollWithCorrectUser() ✓
    - testClosePollWithIncorrectUser() ✓

Img 7. Unit tests for ViewModels and the algorithm, all passed

Automated testing:

The foundation blocks of automated testing are **unit tests** (found in the rankedTests directory). They are great at isolating units (eg functions or methods) from the rest of the system. For this, we need to mock anything that is not strictly under our control (ie the API and network requests). For this a good solution is to use **dependency injection**. This is done by creating a protocol (PollServiceProtocol) that specifies what the service does, and use this in the ViewModels, and then initialize the VMs using the real service. We then can use the real service (PollService) for the app, while using a mock (MockPollService) for testing. We can make sure with this that the system under testing (sut) is not being influenced by unknown factors.

After a closer look there's no testing needed for things such as missing or nil parameters when those are not optional (eg in an API call), Swift in general does a great job of not allowing developers to make these mistakes. For example if I were to try to upload a vote without a title or creator name, it would not compile.

Future roadmap

- Allow users to migrate their initial anonymous account to a permanent one, preferably using sign-in services (eg Google, Apple, or Facebook accounts).
- Set the visibility of polls to only users who have created them or had the IDs shared with them.
- Be able to share polls (by their IDs) with messaging services, as well as multipeer connectivity
- UI tweaks: allow users to change the color of the polls, and display the results in a more user-friendly way (this requires a lot of infographics research)
- Have more testing coverage, especially integration tests. The unit tests in place are good, but they're very artificial, that's when we can use integration tests, to see how it runs when several systems work together (more similar to the real-world cases)
- Release on the AppStore!

How to deploy it yourself

If you're looking for instructions on how to build and run the project yourself, please refer to the README file on the GitHub repo.

Bibliography

- Riker, W. H. (1982). *Liberalism Against Populism: A Confrontation Between the Theory of Democracy and the Theory of Social Choice.* Amsterdam University Press. Pp 29-31. ISBN 0881333670.

- FairVote. (2022, October 28). *Ranked Choice Voting.* Link.

- SBS News. (2013, August 14). *Explainer: What is preferential voting?* Link.

- *Swift - Apple Developer*. Retrieved November 3, 2022, from link.

- *SwiftUI - Apple Developer.* Retrieved November 3, 2022, from link.

- *Atlassian. Gitflow Workflow | Atlassian Git Tutorial.* Retrieved November 3, 2022, from link.

- *UserDefaults | Apple Developer Documentation.* Retrieved November 3, 2022, from link.

- *CoreData | Apple Developer Documentation.* Retrieved November 3, 2022, from link.

- *SQL versus NoSQL: Pros and Cons | DataStax.* Retrieved November 3, 2022, from link.

- *Authenticate with Firebase Anonymously | Firebase Documentation.* Retrieved November 4, 2022, from link.

- B. Mueller. (2017). *OWASP Mobile Application Security Verification Standard*.

- Colvin, T. (2022, January 4). *Is Firebase's Firestore database secure? | Medium.* Link