Marco Alban
Roberto Alvarez
Anton Apostolatos
Leonard Bronner

Cramr

Introduction

Cramr is an iOS application that provides an integrated and easy-to-use platform of students to form study groups, it facilitates and enables student success by fostering an environment of sustained cooperation. In the following pages we will be documenting and detailing the various aspects of the project, written in the Swift programming language.

Quickstart

To download and run the app from the codebase, do the following

- Be a confirmed Facebook tester on developers.facebook.com (currently this only applies to Francois Chaubard)
- Clone the repository
- Open the Xcode project file ('Cramr v2.0/Cramr v2.0.xcodeproj')
- Connect the iOS device.
- Build and run on device, accept permissions when prompted.
- For testing purposes, there are existing session made for CS106A and CS194.

SDKs, APIs and Libraries Used

Parse

Parse is a mobile app platform that handles the app's data management both locally and server-side. The Parse API enabled us to offload the work involved in creating a database backend and sending push notifications thereby enabling us to devote more time to user testing, design-flow, and creating a streamlined user experience.

Facebook

The Facebook SDK is used to handle user registration, login, and invitation to sessions. When a user first opens the app, they are prompted to login through Facebook; once logged in, their account is automatically created and integrated into our server back-end. Using a familiar Facebook login allowed us to create a seamless integration with a registration tool that users have used in the past.

Google Maps

The Google Maps SDK is used to display the location of study sessions as well as allow a familiar outlet for users to set a location. In user tests we established that enabling the user to browse the map to set a location for a study session proved to be intuitive and easy.

CocoaPods

It is important to note that we did not use any CocoaPods in the development of this application. Integrating these CocoaPods into a Swift development environment proved to be very time-consuming and difficult. Ultimately, the marginal benefit of using CocoaPods was outweighed by the complexity of its integration.

General Data Management

Local Data Storage

The **LocalDatastore** class stores important user and application information, including the user ID, the username, and the session ID of the user's current active session. By using the local datastore functionality provided by Parse we can provide a seamless resume from when the app is dismissed and when it is force-closed. In addition, local data storage enables to maintain some limited functionality in the case of no data connectivity.

The session ID is stored in order to provide information to **AppDelegate** as to whether or not the user is currently in a session. This design allows for efficient navigation controller logic. That is, it allows **AppDelegate** to quickly display the

correct view (locked session view versus the master view controller) without having to query the server from application launch.

An instance of the **LocalDatastore** class is stored in the **AppDelegate**. View controllers seeking to access local data can do so by invoking the **AppDelegate** as per MVC architecture.

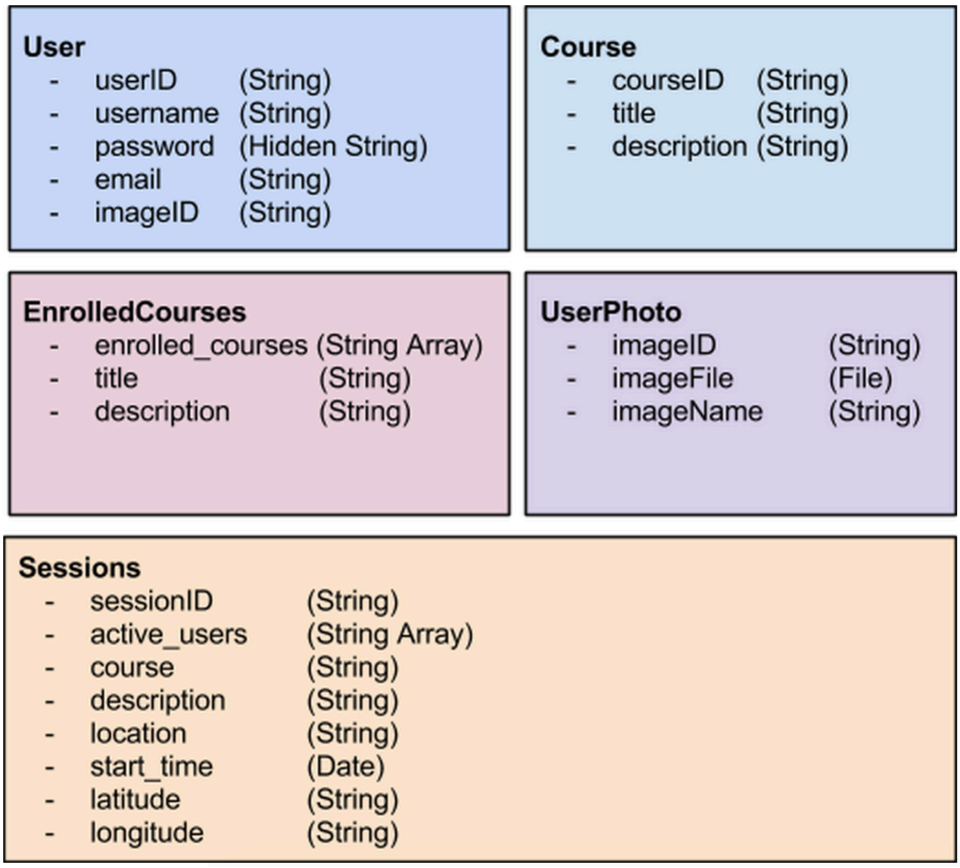
Server-side Storage

The remaining data management is all done within the **DatabaseAccess** class. The class itself contains all necessary functionality to communicate with the Parse server. All interactions with the Parse database, including any queries performed, are performed in background threads. Every function receives a callback function which is run upon task completion. Let's consider the example where the locked view is to be displayed upon starting the app, as the user is currently in a session. The **AppDelegate** calls the pertaining **DatabaseAccess** class to get the specified session's information (namely, **getSessionInfo**), passing in a callback function that instantiates the locked view upon receiving the relevant session information. This general framework applies to every function in the **DatabaseAccess** class.

Similar to the **LocalDatastore** class, an instance of the **LocalDatastore** class is stored in the **AppDelegate**. View controllers seeking to access local data can do so by invoking the **AppDelegate.**

Given that the call to get user pictures from server is the most data-intensive, as well as the fact that a user's picture is rarely going to change, user pictures are cached locally. Thus, a given user picture is requested only if that user has never been seen before. By simply storing ~6KB pictures of users (200 pictures constitute only slightly more than 1MB) the app's quickness and responsiveness increased significantly.

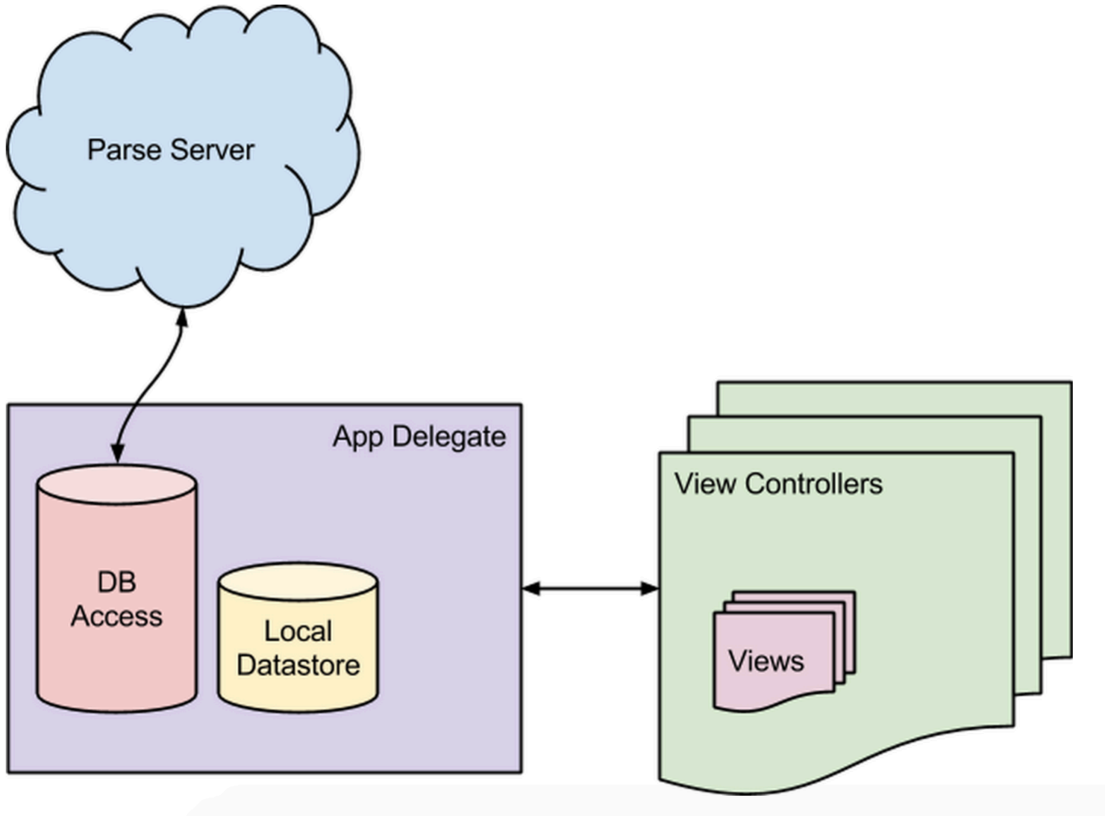
Database Schema



Push Notifications

If the user has allowed push notifications from the App (we register notifications in the **AppDelegate**), inviting friends to a session will send the invited friends push notifications, notifying them that they've been invited to a session. The management of these push notifications is all done through Parse. The most technically challenging aspect was adding robustness for all the possible scenarios that a user could encounter upon receiving a notification. This includes the user having the app open versus having the app closed, the possibility of being invited to a phantom session (i.e. a session where every user has left), and being invited to a session when the invitee is not currently enrolled in that course.

Structural Overview of Code



AppDelegate

The **AppDelegate** class is responsible for interfacing our view controllers with our local datastore (through our **LocalDatastore** class) as well as our parse server (through our **DatabaseAccess** class). Initially, we made parse queries within each view controller, but it led to multiple concurrency issues, and thus we refactored the codebase and abstracted away all of that complexity to within the app delegate as an instance of the **DatabaseAccess** class. In this way, we could make asynchronous calls by including a callback function to be executed upon completion. Similarly, an instance of the **LocalDatastore** class is stored in the app delegate. Both instances are accessed by the application's view controllers directly through the **AppDelegate**.

Additionally, the **AppDelegate** is responsible for the application launch behavior. For example, if a user is in an active session, he or she should be sent to the locked view corresponding to the active session instead of to the 'My Classes' view. Similar

checks are performed for the receiving of session invites (through push notifications) to see whether a user is already logged in, as well as if a user should go to the onboarding view or not.

Login

Logging into the application is handled in the **LoginViewController** class. This class performs two tasks: integrating with Facebook and playing the background login video. The first task required a thorough understanding of the Facebook SDK, including how to properly add the Facebook login button view. This required having our controller serve as a Facebook delegate in order to retrieve the user information and then add it to parse. The video player required instantiating a AVPlayer class and playing the video. Furthermore, this class is responsible for checking if a user has already used the app and then sends them to the onboarding view controller as opposed to the master view.

Master View

The master view is the primary view a user sees when he or she opens the application. It contains a tableview with a list of a user's courses as well as functionality to add courses through the '+' button in the top-right. Courses can be removed by swiping left. The tableview includes refresh ability and is otherwise a fairly standard tableview aside from our custom cells.

The table view contains custom cells (**CustomCourseTableCell**) for each course with an indicator of the number of active users and the number of active study sessions. Tapping on the cells takes the user to the session browser so they can join the one they prefer. If no such session exist, then a '+' icon is shown instead. Tapping on the cells, in this case, takes you directly to the session creation screen.

The challenge with the cells was in querying the Parse database for the user's classes asynchronously in the background. This involved properly updating the

views independently through multithreaded queries to the Parse server, with an extensive use of callback functions.

Onboarding

An important aspect of the application's user experience is the onboarding process. A welcome scrollview introduces the user to the app the first time he or she opens it. We opted to include a walkthrough of the app, with video demonstrations of each major function: adding a course, creating a session, and browsing through existing sessions.

To implement this we used a PageViewController with a page for each major step. The pages display a header text and a video that plays once that specific page has been loaded. The **OnboardingViewController** instantiates an **OnboardingContentViewController** for each of these specific pages. We then included code in our **AppDelegate** to trigger the onboarding upon first use. The major challenge with onboarding was to record short yet sufficiently explanatory videos, as well as how to include and play these videos. To play the videos we made an instance of the AVPlayer class, making necessary modifications for proper aspect ratio and size.

Add Course View

The **CourseListTableView** allows the user to search for and add courses in which they are enrolled. This is done by using a searchable table view where each cell is a course. Once the user selects a row, the course is added to their list of classes (shown in the Master View) and the user is sent back to the Master view.

The course information was acquired by scraping Courserank for class information. By using a regex modification of the user's search query we were able to create an intuitive, responsive, and scalable search mechanism for courses. Finally, we added

UI tweaks such as a magnifying-glass image in the background to make the page feel intuitive.

Session Browser

The session browser view is what we envision to be the most frequented view by the user. Therefore, we invested a heavy amount of time in making sure the experience was seamless and errorless. The session browser consists of three files: the **SessionBrowserViewController**, the **SessionViewController**, and the **SessionContentViewController** (each one called by the previous). The content view is contained in a page view controller within the session browser.

One challenge in creating the session browser was to instantiate the different pages programmatically as opposed to in the storyboard. The other challenge was in quickly loading the content for multiple sessions quickly. The first was overcome by precise passing of session information. The second was handled by caching static session information such as the user profile pictures to not have to make database calls each time. Furthermore, each content view includes a map marked with the location of the session as well as the users location. This was done by integrating the Google Maps SDK. Last, each content view contains a thumbnail of the profile pictures along with the name to give the user a visual and textual representation of who is in the study group.

Session Creation

Creating study sessions is done within the **SessionCreationViewController**. Once the view loads, we perform a few checks to ensure the UI elements are the right color and style, and then we proceed to add a Google map view. The view we present is similar to that of apps such as Uber and Lyft where the user drags a pin on the map to the desired location of the session, enters a description, and then creates the session.

Programmatically, the main challenges involved were properly extracting location information and adding a custom “My Location” button. The creation of this button involved gaining a deep understanding of the Google Maps API. By doing so, we were able to recreate the same functionality as their own “My Location” button but also achieve our desired feel, zoom, and map movement. Lastly, we included error alerts to prompt the user to input the right data.

Session Locked

The session locked view is perhaps the second most important view in the entire application. This name refers to the fact that user is locked to the session once he or she joins, and can only go a different view in the app if they leave the session. Ensuring that a user stay in a session required storing session identification information and ensuring that no matter what state the app is being restored or opened from, a user in a session is sent to the session locked view. The primary motivation behind this functionality is to ensure that a user does not join other sessions, or stay in a session accidentally.

Furthermore, this view presents the same description, room, and a map of the session as the Session Browser View. The major difference is that the locked view allows the user to invite friends to a session. We used the Facebook SDK to integrate a friend picker and Parse to send a notification to the selected friends. Then, we used similar code to the session browser content views to display the users in the study group by displaying circular thumbnails of their profile pictures along with their names, with the addition of the button to invite friends. Finally, we included a background refresh of the page as well as a manual refresh button to allow the user to see who has joined the study group.

Util

The **Util** file contains a number of utility functions and variables used throughout the entire project. This includes the definition of the **cramrBlue** color, utility functions like

getCourseID, which takes a course description like “CS 228: Probabilistic Graphical Models” and returns the course ID “CS228”, and **checkForNetwork**, which is a universally used functions which launches a background thread which, in the case of there being no network connectivity, presents an alert alerting the user of such. The file also includes an **addBlur** function, used to add translucency to subviews throughout our application.

Robustness

A principal consideration we took during each of our programming design decisions was that of robustness. We sought to create an application which would withstand even the most obscure of scenarios.

First, we were careful to make sure that the application state for every user was always consistent with the database state at all times. We made all queries run in background threads in order to not block main thread, only performing certain functions such as segues or table view reloads until those queries were completed (through the use of callback functions). No action that was contingent on information being received or sent to the database was performed until the database had been modified to reflect such a change. As the app grew in scope, size, and complexity this design paradigm allowed for a more reliable application and a more streamlined development process.

Since our application hinges on almost constant communication with the server, it was vital that the application was robust to sudden changes in connectivity. Anytime the user performs an action that requires network access, we check for connectivity before attempting to perform said action. If the connection is nonexistent, then we gracefully handle what would have otherwise have led to an error with informative prompts and alerts.