1. Implementation
   1. Backend integration

The iOS prototype runs on a backend provided by Parse. All data is stored and managed in the Cloud and accessed through the interface provided by the Parse SDK. The following sections will give a brief overview about how the framework is integrated and used and explain some its most useful features.

* + 1. Connecting to the backend

To hook up the app to the Parse backend, the following steps had to be completed:

* Sign up for Parse and create an app within the Parse client
* Download the Parse SDK and include it in the Xcode project
* Add additional iOS libraries required for the Parse framework to work properly
* Register the client with the Parse app by running the following code upon application lauch:

[Parse setApplicationId:@"ltQB4UH8RtuQ84RTJOWg16IfJh0fojlzrYEbwwUr"

clientKey:@"Hi4lrAsfSq0iWDi6npeYMlgrmL65l5iWFtoKl5Ef"];

* + 1. Storing Data and Building the Data Model

Once conncected to the backend, the client has access to all stored data and is able to create, retrieve, modify and delete datasets. Data objects are represented by the PFObject class. Properties are stored as key-value pairs. Supported value types are: String, Number, Boolean, Date, File, GeoPoint, Array, Object, Pointer and Relation. As indicated through the Pointer and Relation type, the framework supports relational structures and provides a convenient interface for relational queries.

The data model can be created through a user interface on the Parse website. However, data management within the Parse framework is schemaless, which means that it is not required to specify the schema of a class or even the class itself before creating instances of it. If an object with a certain class name is sent to the server and there are currently no other stored objects of that class, Parse will lazily create a new class for the received object.

The following code snippet illustrates the process of adding an object to the persistent storage:

PFObject\* messageObject = [PFObject objectWithClassName:@"Message"];

[messageObject setValue:@"Hello" forKey:@"messageText"];

[messageObject save];

In addition to the simple saving function, the frameworks offers a variety of additional functions for saving objects such as saving in background, running a callback function or block upon server response or, if the device is currently offline, caching the object and saving it as soon as network connection is restored.

* + 1. User Creation and Management

The framework also offers extensive functionality to create and manage user objects. For this purpose, the PFUser object is used. User accounts can be created by sending a PFUser object with a username, an Email address and a password to the server. The server will verify the uniqueness of the Email address and username and send out an Email asking the new user to verify their Email address. Once verified, user may sign in with their credentials. After successfully signing in, the user object will be cached and globally available through the [PFUser currentUser] function. Unless the client specifically signs out the current user by calling [PFUser logout], the user will remain signed in even after closing and re-opening the app.

* + 1. Retrieving Data

To query for stored data, the PFQuery class is used. A query can retrieve a single object through the object's unique objectId property, or search for objects that match one or several specified conditions and return those objects in an array. The following example shows a simple query to retrieve offers that match a certain category.

PFQuery\* query = [PFQuery queryWithClassName:@"Entry"];

[query whereKey:@"entryType" equalTo:@"offer"];

[query whereKey:@"category" equalTo:@"Babysitting"];

[query findObjectsInBackgroundWithBlock:^(NSArray \*objects, NSError \*error) {

if (objects && [objects count] > 0) {

// found something! Display the results to the user

}

}

The PFQuery class offers a wide range of functions to constrain searches and sort and filter results. This was particularly helpful for implementing the search feature in which not only entry titles, but also their corresponding descriptions should be included in the search. For such complex queries, several subqueries can be linked with a disjunction using the following function:

+ (PFQuery \*)orQueryWithSubqueries:(NSArray \*)queries;

It turned out that the PFQuery class was also able to take care of filtering results by a given geolocation and a corresponding radius. This is exactly what was needed for the search feature if a user wants to limit their search to a certain area. The user input representing an address, ZIP code or city or a combination of those will be geocoded into latitude/longitude coordinates and, along with the user-specified search radius, can be set as a constraint for the query using the following function:

- (void)whereKey:(NSString \*)key nearGeoPoint:(PFGeoPoint \*)geopoint

withinMiles:(double)maxDistance;

* + 1. Push Notifications

The Parse Frameworks supports push notifications in a very easy-to-use way. Through the web interface of Parse, custom push notifications can be sent to all devices that installed the application (provided they gave permission to receive notifications), for example to inform users about an update. Additionally, the framework allows for client-side notifcations, a feature that is used within the app to let users know that they received a new message. This is easily achieved through the following code:

PFQuery \*pushQuery = [PFInstallation query];

[pushQuery whereKey:@"user" equalTo:self.recipient];

PFPush \*push = [[PFPush alloc] init];

[push setQuery:pushQuery];

[push setMessage:@"You received a new message!"];

[push sendPushInBackground];

* 1. Data Model

[TODO class diagram of model]

* 1. View Hierachy

The root of the app's navigation hierachy is represented by the SZNavigationController, a subclass of UIViewController containing two essential components: The hidden menu and a UINavigationController serving as the root for all added subviews. The hidden menu, represented by the SZMenuVC class, is designed as a singleton and subclasses UIViewController. It contains a UITableView to layout and skin menu cells and arrange them into sections. The menu uses delegatation, a common design pattern in iOS applications, to inform its delegate when a user selected a cell. Menu cells are assigned a corresponding class name that is passed on to the menu's assigned delegate to perform the view switch requested by the user. The menu protocol declares the the following function that its delegate has to implement with the appropriate action:

@protocol SZMenuVCDelegate <NSObject>

- (void)menu:(SZMenuVC\*)menu switchToViewControllerWithClassName:(NSString\*)className;

@end

The SZNavigationController encapsulates the core navigation logic and serves as the delegate for the SZMenuVC. When it receives a message from the menu to that the user requested to navigate to a different section of the app, it creates an instance of the according class, sets this class as the root view controller of its main view container and hides the menu:

UIViewController\* vc = [[NSClassFromString(className) alloc] init];

[self.mainViewController setViewControllers:[NSArray arrayWithObject:vc]];

[self slideInMainViewAnimated:YES navigationType:SZNavigationMenu];

View Controllers that serve as the root of one of the app's sections will display a "Menu" button on the top left of their navigation bar. Tapping this button will slide the current view to the right of the screen, revealing the underlying menu. Tapping the button again moves the view back into place and hides the menu. However, once a user navigates one or more levels down in a section's view hierachy, the "Menu" button will be replaced by a "Back" button. This is one of the most common interaction paradigms in iOS applications and must be preserved in order to ensure proper usability. On the other hand, it would be inacceptable to require users to navigate all the way back to the top of a section's view hierachy to access the menu button. Therefore, the menu is made available globally through a swipe gesture. By swiping across the screen to the right, users can make the menu appear at any given time, regardless of how deep in the view hierachy they currently are. This is realized through a UIGestureRecognizer set on the SZNavigationController's main view container. The gesture recognizer detects swipe gestures and informs the SZNavigationController, which takes control of sliding the main view in or out accordingly.

The main challenge with implementing the menu behaviour was the special case of modal view controllers. Normally, new views are presented to the user as a new level in the view hierachy, meaning they will be pushed on the current view controller stack and visually slide in from the right. If a user taps the back button, the view will slide out to the left and popped from the view controller stack, bringing the user back up in the view hierachy. In some cases, users should be presented a view controller that is not really part of the current view hierachy but instead encapsules its own view hierachy for which it serves as the root view. Often this applies to distinct user flows that require a "Cancel" option. With regard to this project, a good example for this scenario is the "My Listings" section. Assuming the user is currently viewing a list of their current offers, selecting one of those offers will open up the offer's detail view. The user may go back to the list view by simply tapping the "Back" button at the top left of the navigation bar. In the list view, a button titled "Create New Offer" lets the user create a new offer. This action brings up the view for creating a new offer, which consists of several steps, all of which are represented by a separate view controller that will be pushed onto the view stack. Assuming the user already reached step 5 before changing their mind and wanting to cancel the offer creation, it would require the user to tap the "Back" button 5 times before reaching the list view again. This is where modal view controllers come into play. Modal view controllers typically slide in from the bottom, leaving the underlying view controller in place and introducing their own view hierachy. They usually display a "Cancel" button at the top right of their navigation bar which lets the user dismiss the currently displayed view hierachy and instanly go back to the underlying view hierachy. The user flow for creating a new offer or request is implemented in such a way. While modal view controllers usually only allow for navigation within their own hierachy, in the special case of this project it is necessary to slightly vary from this standard to ensure a consistent user experience and keep the hidden menu accessible at all times. This could be realized by using an additional instance of the SZNavigationController class to serve as the container of modal view controllers and offering those controllers the same swipe functionality. Since the menu class is handled as a singleton, setting its delegate to a new SZNavigationController instance and assigning its view to that instance resulted in the issue that once a modal view controller was dismissed, the underlying view controllers were suddenly missing the menu. After many failed attempts to solve this issue, which always resulted in some kind of odd behaviour by producing unexpected side effects, the solution that actually worked was so simple that it was almost too good to be true. The UIViewController class has a built-in function that is called whenever a controller's view appears on the screen, regardless of whether it is displayed for the first time, re-appearing after being hidden by another view controller or popping back up in the view hierachy. Overriding this method in the SZNavigationController class ensures that the menu view will always be hooked up to the SZNavigationController instance that is currently being displayed to the user:

- (void)viewDidAppear:(BOOL)animated {

[self addChildViewController:[SZMenuVC sharedInstance]];

[self.view insertSubview:[SZMenuVC sharedInstance].view atIndex:0];

[[SZMenuVC sharedInstance] setDelegate:self];

}

* 1. Code Architecture

The code is generally designed to encapsulate logic into distinct modules and minimize unneccsesary interdependencies between classes. Every screen is represented by it's own UIViewController subclass which only holds references to the information it really needs. For example, the SZSearchVC implements the search screen in which the user can input keywords, select a category and specify a search area. The SZSearchVC class takes care of creating and laying out its user interface and handling the user input. The SZSearchResultsVC on the other hand is responsible for displaying the search results. It needs to know what to search for, but giving it a reference to the SZSearchVC would increase interdependency between those two classes. Therefore, the SZSearchVC takes care of validating the user input, wrapping up all search parameters into a PFQuery object and passing this query into the SZSearchResultsVC. For this purpose, the SZSearchResultsVC interface defines the following public initialization method:

- (id)initWithQuery:(PFQuery\*)query;

The SZSearchResultsVC can now simply run the query and present the search results to the user without needing to know about the specific search parameters itself and without keeping a reference to the SZSearchVC class. Thus, the SZSearchResultsVC is designed flexibly enough to also be used within the browsing feature, in which the user only selects a specific category (or browses within all categories at the same time) but doesn't add any further constraints to the query that is passed into the SZSearchResultsVC.

Sometimes objects need to be accessible throughout several different view controllers and may be modified by each of those controllers. A good example for this is the case in which a user wants to create a new listing. As mentioned before, this is done in several steps, all of which have their own logic encapsulated into a distinct view controller. Upon tapping the "Create New Offer" or "Create New Requst" button, a new instance of PFObject is created to store all the information that the user wants to specify for this listing. Instead of passing the object from one View Controller to the next, the object is assigned to the SZDataManager, which is designed as a singleton class and therefore globally accessible from everywhere in the code. Once a listing is complete and ready to be posted to the server, any class may call the following public function on the SZDataManager singleton:

- (void)saveCurrentEntry:(void(^)(BOOL finished))completionBlock;

The SZDataManager will dispatch a completion block when it has successfully saved the listing to let the calling class handle whatever it wants to do when the saving process is done. In this case this would be removing the spinning wheel that was indicating the ongoing saving process for the user and navigating back to the list view that is displaying all of the user's current offers or requests.

Outsourcing the actual saving process to the SZDataManager results in the benefit of having the saving function available in one single place. Therefore, when a user decides to edit for example only the title of a specific listing, which is done in Step 2 of 5, the saving function can be called directly from Step 2 without requiring the user to go through all of the other steps and without having to implement the save functionlity in every single of the five view controllers.

Data that needs to be globally available to all classes for convenience is stored in a class named SZGlobalConstants, which is included in the project's header prefix and therefore by default available in all classes without the need to explicitly import it. This class defines certain enumeration types that are used in different places of the code and offers a few class methods that will return values used for skinning the app. For examlpe, setting any random label used anywhere in the code to the app's specific font in a certain font weight and giving it the typical dark petrol color can be done like this:

[textLabel setFont:[SZGlobalConstants fontWithFontType:SZFontSemiBold size:16.0]];

[textLabel setTextColor:[SZGlobalConstants darkPetrol]];

Another convenience class that offers some more complex functionality used in different places of the app is the SZUtils class. It mainly takes care of simple tasks such as converting an NSDate object into an NSString that represents the date formatted in a humanly readable way, but also offers some more complex functionality like taking an array containing numbers from 0 to 5 that represents a user's review profile and returning a view that display's the average of those review scores in an typical "star view".

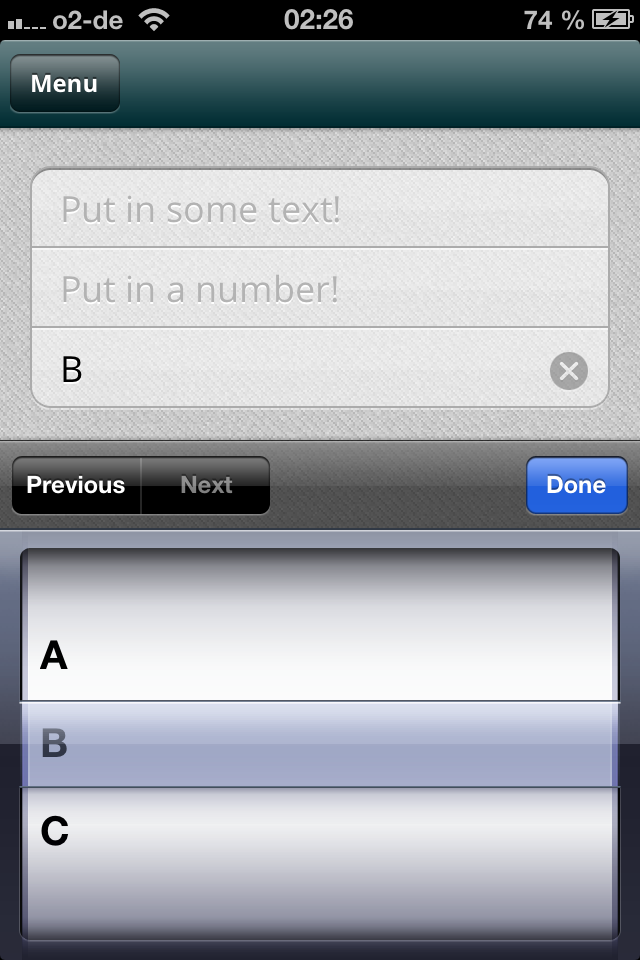
* 1. Custom UI Elements

There are several UI elements that used within different view controllers that should always have the same behaviour and look the same way. Sometimes they are just generic control elements offered by the iOS SDK that need to be skinned to integrate into the app's visual design, but in other cases they need to incorporate more complex control logic to store and process user input. For this purpose, creating custom classes that subclass either the UIView class or other specific UI classes was the right approach. The SZButton, a subclass of the UIButton class available in the UIKit of the iOS SDK, will generate a button that fits into the app's visual design with the following initialization method:

- (id)initWithColor:(SZButtonColor)color size:(SZButtonSize)size width:(CGFloat)width;

The color and type properties are defined by enumeration types to ensure consistency. Available button colors are petrol (SZButtonColorPetrol) and orange (SZButtonColorOrange), available button sizes are heights of 24px (SZButtonSizeSmall), 32px (SZButtonSizeMedium), 40px (SZButtonSizeLarge) and 52px (SZButtonSizeExtraLarge). The button width is fully flexible and specified by the width parameter.

An example for a more complex UI element is the SZForm class. An SZForm instance is an abstracted component that lets users input data in different ways. It is displayed as a view element containing one or more fields in which the user can enter information. Each of these fields can be configured individually by using an SZFormFieldVO to customize the way this data is supplied. For instance, one field may be behaving like an ordinary text field that uses the default keyboard, the next field may be requiring a numeric value that uses a number pad instead of the keyboard to supply the input while another field is requesting the user to pick from a pre-defined set of options and uses a UIPickerView to let the user choose one of these options. A form configured in such a way would look like this:

The corresponding form component can be easily created using this code:

SZForm\* form = [[SZForm alloc] initWithWidth:290.0];

SZFormFieldVO\* textField = [SZFormFieldVO formFieldValueObjectForTextWithKey:@"text"

placeHolderText:@"Put in some text!" keyboardType:UIKeyboardTypeDefault];

SZFormFieldVO\* numberField = [SZFormFieldVO formFieldValueObjectForTextWithKey:@"number"

placeHolderText:@"Put in a number!" keyboardType: UIKeyboardTypeDecimalPad];

SZFormFieldVO\* pickerField = [SZFormFieldVO formFieldValueObjectForPickerWithKey:@"picker"

placeHolderText:@"Pick!" pickerOptions:[NSArray arrayWithObjects:@"A", @"B", @"C", nil]];

[form addItem: textField showsClearButton:YES isLastItem:NO];

[form addItem: numberField showsClearButton:YES isLastItem:NO];

[form addItem: pickerField showsClearButton:YES isLastItem:YES];

Internally, the SZForm class takes care of various tasks such as retrieving and displaying the correct background view for the field depending on whether it is on top, in the middle or at the bottom of the form, configuring the text fields with their placeholders, input types and visual properties. With the help of an open source component called "BSKeyboardControls"[[1]](#footnote-1), it creates a toolbar for convenient switching between fields through "Previous" and "Next" buttons and dismissing the input view with the "Done" button. When selecting a field that is positioned in the lower part of the screen and would therefore be hidden by the input view, the SZForm also ensures that the screen will be scrolled up accordingly to make the respective field visible for the user. User inputs are stored in an NSDictionary (using the keys provided by the SZFormFieldVOs) that can be accessed through a public property by the class that is using the form. Additionally, the SZForm can be assigned a delegate and defines a protocol with various functions that delegates may implement if they wish to be informed about certain events, for example to validate input after a user tapped the "Done" button.

* 1. Caching on the device

While features like searching or browsing for listings obviously need a working network connection, there are certain sections of the app that wouldn't necessarily require the user to be online in order to be functional. Reading old messages, viewing one's profile page or accessing a list of one's own posted listings for example are examples of scenarios that would make sense to have available even in an offline environment. However, all data is stored in the cloud by default and therefore even trivial information such as the current user's name is only accessible when the device is connected to the internet. To solve this problem and offer users some useful offline functionality while at the same time increasing performance, certain data is cached on the device.

The SZDataManager class takes care of all caching tasks and uses a .plist file stored in the app's default documents directory to save data to the disk and structure it in a way that makes it easy to retrieve and convert back into a format that the code can process. Taking the example of caching received messages, locally stored message objects are stored as a nested dictionary. A message itself is represented by a dictionary using keys like "messageText" and "fromUser" or "toUser" to store values. The message dictionaries are in turn stored in a dictionary that uses timestamps of when they were received as unique keys. This makes it easy to determine the newest of the stored messages and also allows for a fairly trivial sorting algorithm to arrange the messages appropriatly before passing them on to the class that will take care of displaying them to the user. The SZDataManager offers the following function to check for new messages:

- (void)checkForNewMessagesWithCompletionBlock:(void(^)(BOOL finished))completionBlock;

Within this function, the SZDataManager will determine the newest of the stored message keys, transform it into an NSDate object and run a query for all messages that were received after that time. If the query doesn't return any results, the local store is up to date, if there were new messages received, they will be added to the local store. Finally, the completion block that was passed into the function will be executed. Depending on which class is calling the message check function, this block could do different things. If it's called by the SZAppDelegate upon application launch, it would be informing the user about new messages, if called by the view controller that's displaying a list of all message threads (grouped by users), it would simply be updating its view accordingly.

1. https://github.com/simonbs/BSKeyboardControls [↑](#footnote-ref-1)