VIA UNIVERSITY COLLEGE

Project Report
Danish Aesthetics' iOS Application

Date: December 9 2014
Mikkel Cortnum Poulsen – 171460
Janis Bruno Ozolins – 166732

Supervisor: Lars Bech Sørensen – LBS

Table of Contents

Table of Figures

Table of Code Snippets

Abstract

The objective of this project, is to build an iPhone Application, which can use the content from the already existing website, present it in a user friendly and interactive way and allow users to track their progress in the gym. The application was built as a very user friendly MVC application utilizing many of the well known and renowned iOS elements, such as the Tab Bar, Table Views, and Navigation Bars. In the end Apple's new programming language, swift, complicated the process and the tracking feature was not implemented. However, the objective was achieved to the fullest, by using their already existing responsive website directly in the application it is even done in a way that complies with Danish Aesthetics image and brand. All in all, the outcome of the project was very satisfactory and future work will ensure that the second objective is also fulfilled.

1. Introduction

This project is going to revolve around the new iPhone Application for Danish Aesthetics IVS. Danish Aesthetics already has a solid website, with many monthly active users. This website provides users with articles about nutrition and dieting along with free workout/weight lifting routines. The company wishes to extend their reach by providing platform where their users can make their workout routines personalized, trackable, easily available and a place where users can find peer inspired relevant information. For this purpose, they want an iPhone Application.

The segment of users are both genders in the age group from 16 to 40 years old and have active lifestyle which involves exercising in gym, bodybuilding, weightlifting and powerlifting. A common person in this target group is extremely dedicated to achieve maximum output from the time spent exercising thus developing his own methods of workout or researching for information from other peers.

The objective of this project, is to build an iPhone Application, which can use the content from the already existing website and present it in a user friendly and interactive way. The users should also be able to create their own workouts and track their progress with these workouts. This tracking feature will require a separate web API and database.

1.1 Business Model

The outcome of this project will be an iPhone application, which is a requirement from Danish Aesthetics. The application will present users with premade workout routines, that the users can follow. What this app will do to differentiate itself from other fitness apps on the market, is that it will allow it's users to track their progress. One of the requirements from Danish Aesthetics is, that the app must provide the users with the ability to track their progress in form of weight, repetitions, and sets.

Another requirement from Danish Aesthetics is that, the application must be able to pull content, in the form of workout routines, from their already existing website. This website is built using the WordPress content management system (WordPress CMS). The company wants to keep using WordPress and the MySQL database that goes along with it. Therefore, the application needs to be able to communicate properly with both the WordPress CMS and the MySQL database.

Alongside this iPhone application, Danish Aesthetics are also working on their new webshop. This webshop is also integrated with the WordPress CMS for managing users and user profiles. These users must also be able to login on the iPhone application and vice versa. Therefore, this requirement also needs to be taken this into consideration throughout the analysis and design of the application.

A fourth requirement from Danish Aesthetics is that, the users must also be able to create their own workout routine, if they do not wish to follow a pre made workout routine. Other than these four requirements, the developers are given free reign to make changes based on user interviews and their own experience and knowledge.

1.2 Introduction to Workout Terminology

This section will briefly explain the basic terminology that will be required when reading the rest of this report. It will cover terms such as sets, repetitions, rest, supersets and more. Feel free to skip this section, if you feel like you know all of this already.

A workout routine is thought of as several different workouts put together to form a workout routine or program that one follows for a given period of time.

A workout is thought of a series of exercises put together to form a continuous effort in the gym. A workout can consist of exercises for one or more body parts, it does not necessarily have to be only one bodypart per workout.

A rest day is a day off from working out.

There are two different types of workout routines; a split and full body workout. **A full body workout** is, as the term might suggest, a workout routine where the entire body is activated during each separate workout. **A split** on the other hand is when each bodypart is activated separately. Even in splits it is common that more than one body part is activated during the same workout. A common 3-day workout split might look something like this:

Day 1 - Chest & Back
Day 2 - Rest Day
Day 3 - Quadriceps & Hamstrings
Day 4 - Rest Day
Day 5 - Shoulders & Biceps & Triceps
Day 6 - Rest Day
Day 7 - Rest Day

An exercise consists of sets, repetitions and rest time. **A repetition** is moving the weight or machine from the start position to the end position and back to the starting position. **A set** is then a given number of consecutive repetitions. **Rest time** is defined as the amount of time to be rested between each set. Sets and repetitions for a given exercise can be written as 3x12; meaning three total sets of 12 repetitions.

It is possible to **superset** exercises. To superset an exercise means that instead of resting when the first exercise's set is finished, a set of the next exercise is completed directly after the first exercise's set is finished. Only after a set has been done for both exercises does the rest time begin. It is possible to superset more than two exercises, this is often referred to as a giant set.

Weight can both refer to the weight that is put on the barbell, machine or the weight of the dumbbells. It can however also refer to body weight. Throughout this report, if the weight in question is body weight, the term body weight will be used, all other uses will refer to the first definition of weight.

2. Analysis

The iPhone applications main goals are: to help motivate users for workout and provide a resource of information and inspiration for a workout routines, which are made by professional athletes or

shared by other users; allow users to create, organize and manage their workout routines and keep track of them.

In order to determine exactly how the app should work and what it should deliver to the users, a short questionnaire was distributed to potential users. The questionnaire got 30 responses and of those 30 only 3 were unusable, meaning the responders did not work out. One key takeaway from the survey was, that 37% of the responders followed a workout routine of their own making and only 20% followed a workout routine they did not make themselves. This can either mean that a lot of people change the workouts they find to make them their own or that they actually make them from scratch. Either way, the app needs to support custom workouts created from scratch and the ability to modify an already existing workout. Furthermore, from the responders that made their own workout routines, when asked what they emphasized when creating their workout routines, half had emphasis on progression. Another key takeaway from the survey was that, when asked what the users where tracking in the gym or wanted to be able to track the answers were very consistent. Almost 100% of the responders answered weight on the bar, repetitions and sets with around 40% also answering rest between sets. This means that the four key parameters to track in the app will be progress in weight on the bar, repetitions, sets and rest between sets. The full spreadsheet with responses can be found in Appendix A - Surveys.

The following requirements need to be met for the application:
Non-Functional

- Application needs to be an iPhone Application
- Application must work flawlessly with many users
- Core data must be accessible across Web and Mobile platforms
- Application must bring top notch user experience

Functional

- Users should be able to use already existing workout routines.
- Users should be able to create their own workout routines.
- Users should be able to edit both types of workout routines.
- Users should be able to track their progress with a certain workout routine.
- Tracking parameters: Weight on the bar, repetitions, sets and rest between sets.
- Users should be able to view statistics about their past workouts.

Nice to haves (All functional)

- Users should be able to share their workout routines.
- Users should be able to view other people's profile (public vs. private profiles)
- Users should be able to see a video tutorial of a certain exercise.

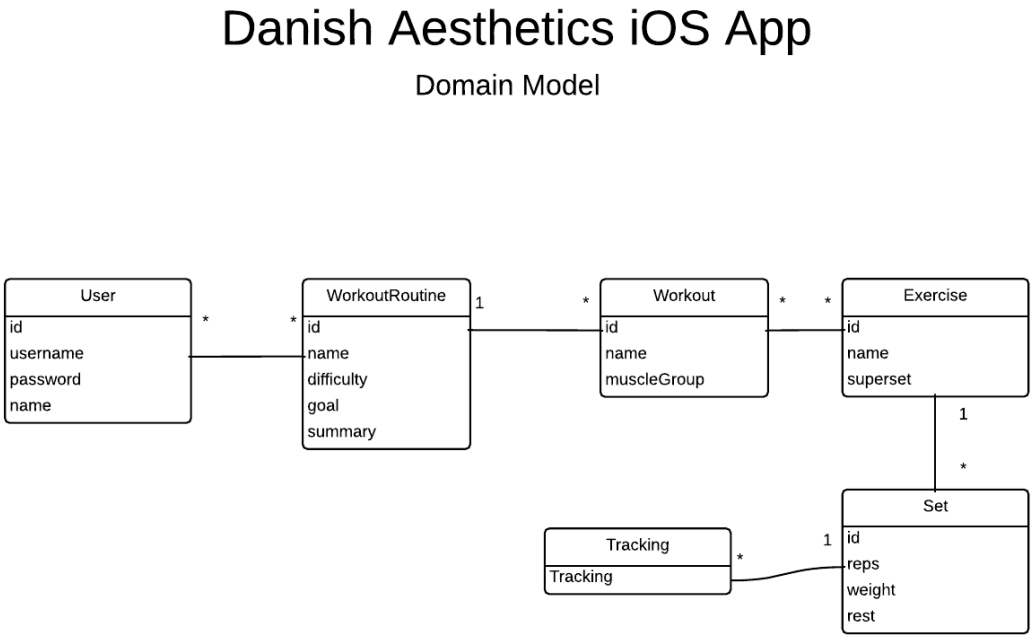


Figure 1 - Domain Model Diagram

The Domain Model diagram perfectly depicts the model that will be used for the application. Each user will be able to have several workout routines and several users may be able to follow the same workout routine. Workout routines in turn consist of at least one workout, which in turn consists of at least one exercise. Exercises have at least one set and sets should be trackable in order to meet to functional requirements of the project. Here the "Introduction to Workout Terminology" section from the introduction should be of great help to understand the domain model.

As the Domain Model diagram aboves perfectly depicts the model of the application, the Use Case diagram below perfectly depicts the main use cases available to any user of the app. As seen in the diagram, a "user" is described as a guest until they create a user account. This means that until a user account is created, the user will not be able to use the application for anything else than creating a user account.

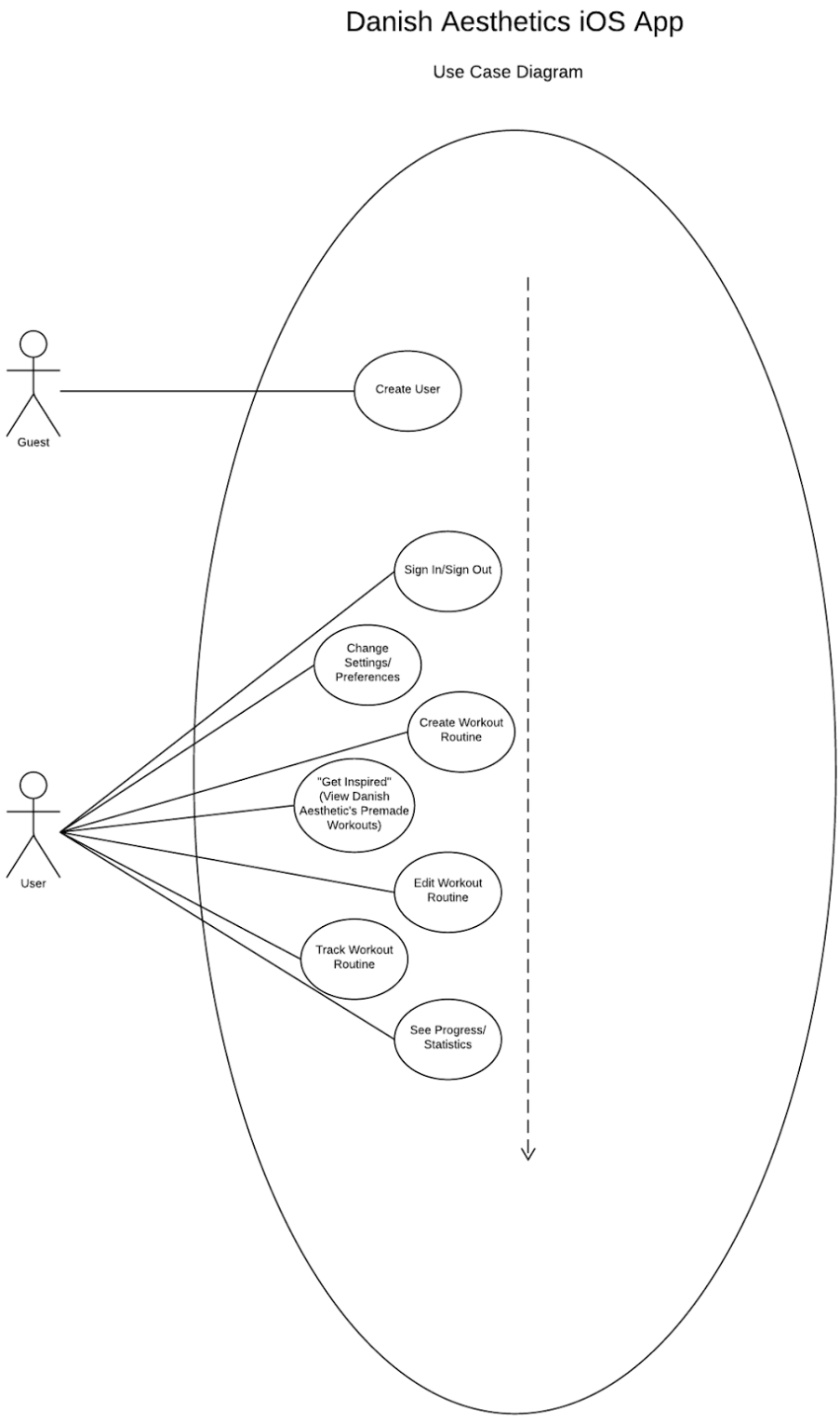


Figure 2 - Use Case Diagram

After a user has been created, they are able to use the application in its entirety. They will for example be able to create their own workout routines. A full description of this use case, along with an activity diagram clearly depicting the process can be found just below.

Use Case Name	Create Workout Routine
Use Case Purpose	The purpose of this use case is to allow user to create a new workout routine
Preconditions	User has launched the application
Postconditions	Workout routine is stored on the app
Limitations	Data is not simultaneously updated in the API database
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone.

Figure 3 - Create Workout Routine Use Case Description

Basic flow

- User navigates to Workout Routine screen
- User presses “Add new Workout” button
- User enters the name of the workout routine
- User additionally enters difficulty and workout type
- User presses “Done” button
- User is navigated to “Add Workout action”

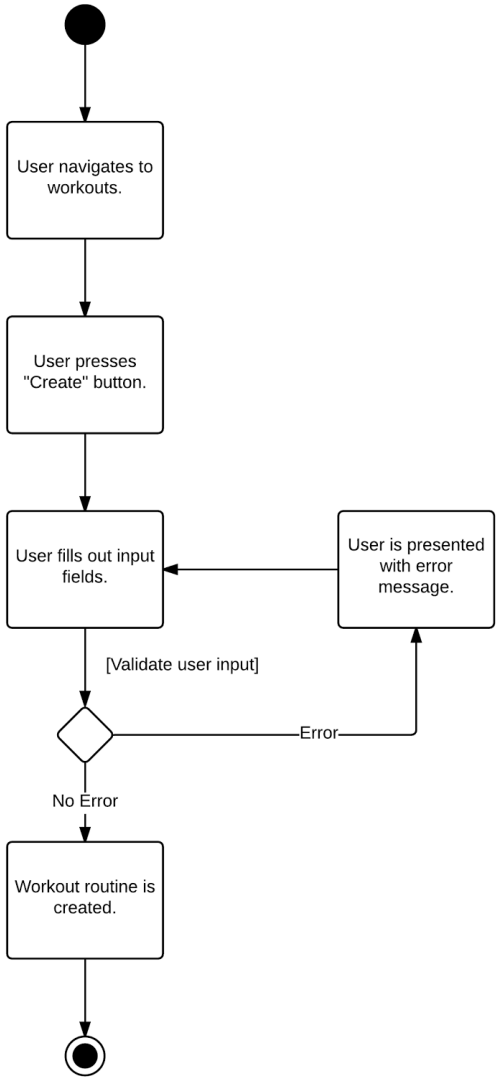


Figure 4 - Create Workout Routine Use Case Diagram

The use case descriptions for all use cases along with activity diagrams for each use case can be found in the appendices section of this report, more specifically in Appendix B - Use Cases.

3. Design

3.1 System Architecture

Because Danish Aesthetics has an already existing website running the WordPress CMS, the application should be able to pull data from this website as one of the main requirements state. However, it would be very impractical to use the database that goes along with the WordPress CMS for storing all the users' data from the iPhone application. Therefore, a system architecture design decision was made, to only use the "legacy" database for pulling content from. All data created by users in the application, will be stored in a separate database that is accessible through a web API. This will also make the users' data available on the website, if Danish Aesthetics at some point

wishes to extend their website with features from the workout application. The reason it would be impractical to use the existing "legacy" database for storing user data, is that the database was built to support the WordPress CMS related content, therefore creating a web server along with a new database, will allow for much easier implementation of new features and migration of data, if necessary. Also the "legacy" database is used for Danish Aesthetics' webshop and it contains sensitive and crucial data for the business daily operations. The overall system architecture is depicted in the figure below.

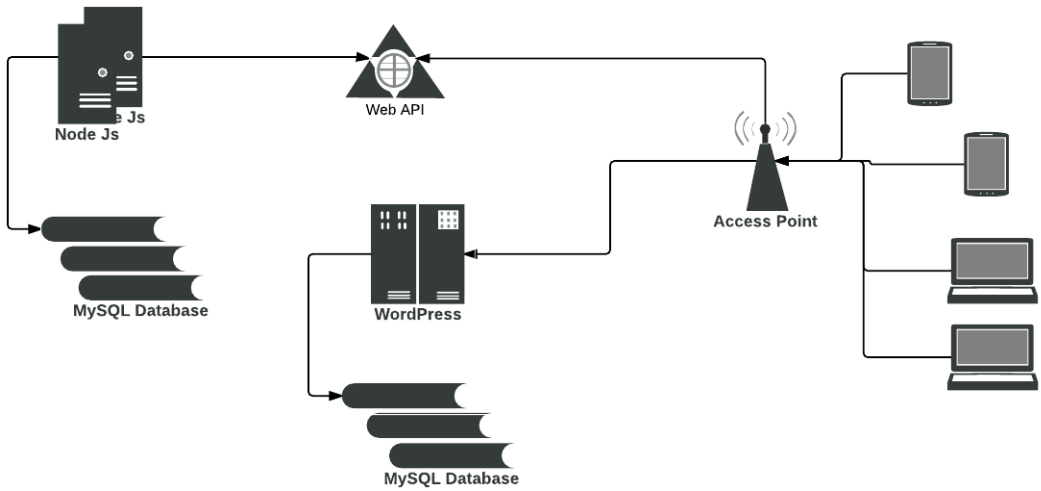


Figure 5 - System Architecture

In the figure above its clearly visible that clients can use either iPhones or personal computers to access content from the Web API and at the same time from the WordPress CMS. The access to the Web API for personal computers is not in the project's scope.

3.2 Programming Language Choice

When making an iPhone application one major decision is to decide which programming language to use. Until very recently, this was not a tough decision, seeing that iOS development was only possible to do in one language; namely Objective C. Another solution was to program it in Ruby or C# and have it compiled into native ARM machine code, that would run on the iPhone, by various toolchains available online[1]. The former solution is often very costly (Xamarin is roughly $1.000 pr. year.) Some of the advantages of using programming languages that are not supported by Apple, is the possibility to develop cross-platform native applications. Cross platform applications allow a better maintainability of code and sharing of the application logic across platforms, which could benefit in further development. However, seeing as the main requirement for this project is to develop an iPhone application these benefits do not apply. The drawback of using programming languages that are not supported by Apple, is the lack of documentation and accessibility of native iOS functionality and elements, which tend to lead to the use of external libraries. The use of external libraries can lead to overhead with obsolete code and performance issues. These drawback

[1] One such tool is Xamarin - http://xamarin.com/

greatly outweigh the advantages; seeing that there are none. Which is why, this option was easily eliminated.

Then during the summer of 2014, Apple introduced a new programming language called Swift. Swift is as perfectly described by Apple:

"*Swift is a new programming language for iOS and OS X apps that builds on the best of C and Objective-C, without the constraints of C compatibility. Swift adopts safe programming patterns and adds modern features to make programming easier, more flexible, and more fun. Swift's clean slate, backed by the mature and much-loved Cocoa and Cocoa Touch frameworks, is an opportunity to reimagine how software development works*."[8]

As Apple says themselves: "*Thanks to this groundwork, we can now introduce a new language for the future of Apple software development*."[8] It is for this reason, along with a multitude of others that will be described in just a second, that Swift has been chosen as the programming language for this project.

One of the benefits using Swift over Objective-C is the efficiency of writing code. For example reversing an array of strings in Objective-C would look like in Code Snippet 1 below:

```
NSArray *myArray =  [NSArray arrayWithObjects: @"Swift", @"Objective C",@"C#",
nil];
NSArray *reversed = [[myArray reverseObjectEnumerator] allObjects];
```

Code Snippet 1 - Objective C: Reverse Array

The same code in Swift would be written as Code Snippet 2:

```
var myArray = ["Swift","Objective C","C#"]
var reversed = myArray.reversed()
```

Code Snippet 2 - Swift: Reverse Array

First of all from the examples above we can see how Swift is more efficient just by using generic types - we do not have to declare the type of the array and furthermore we can make the use of the standard array library method reversed(), which performs the reversion.

All code is written in Xcode, an application to write code in Objective-C, C and Swift, much like Visual Studio for Visual Basic, C# etc.

3.3 Design Pattern

The overall design for the application will follow Apple's version of the widely popular model view controller design pattern. This version of the MVC pattern does does not allow the model and the views to communicate, every communication between view objects and model objects are mediated by controllers.

"*A controller object acts as the intermediary between the application's view objects and its model objects. Controllers are often in charge of making sure the views have access to the model objects*

they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the life cycles of other objects."[19]

Apple aptly describes this untraditional MVC design pattern in the two figures below, taken from their own documentation. Even though the documentation is written for Objective-C and is from January 2012, it still applies for today's swift applications. [20]

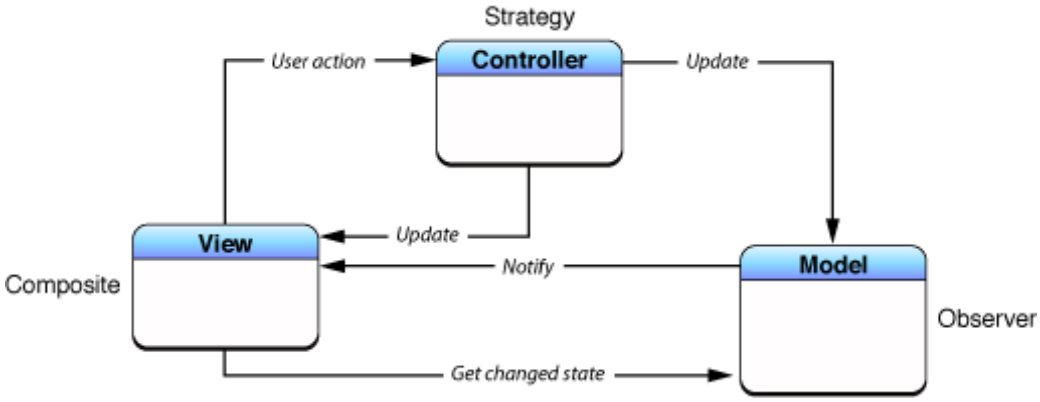


Figure 6 - Traditional MVC Design Pattern

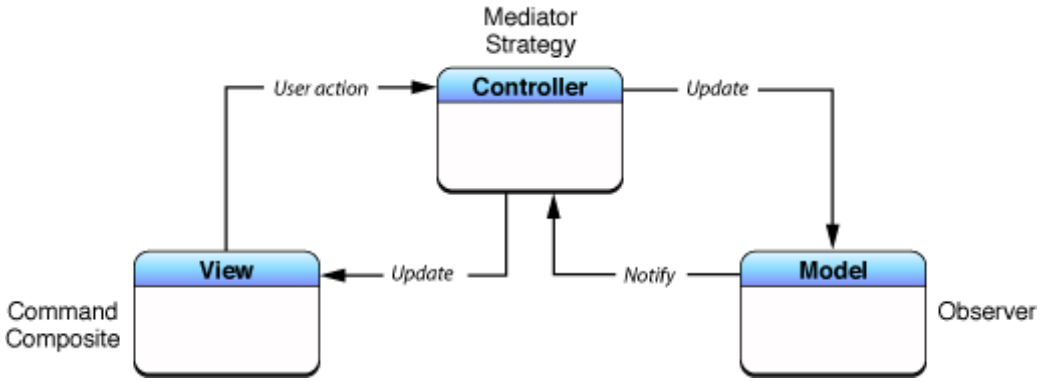


Figure 7 - Cocoa (Apple) MVC Design Pattern

Apple also relies heavily on view controllers, which combine the roles of a view and a controller. For example, a typical view controller could have an outlet referencing a button or table on the view. This outlet allows the view controller to manipulate information input and displayed on the view. Similarly, the view controller can have actions that are called when a user presses a button or swipes across the screen. Apple describes view controllers like so:

"*View controllers are traditional controller objects in the Model-View-Controller (MVC) design pattern, but they also do much more. View controllers provide many behaviors common to all iOS apps.* "[21]

These view controllers are key, when working with storyboards, where a view does not have an actual class, more on this in the "Graphical User Interface" chapter under "Storyboard and Views". How the MVC design pattern will be utilized in this application can be viewed in the UML class diagram below.

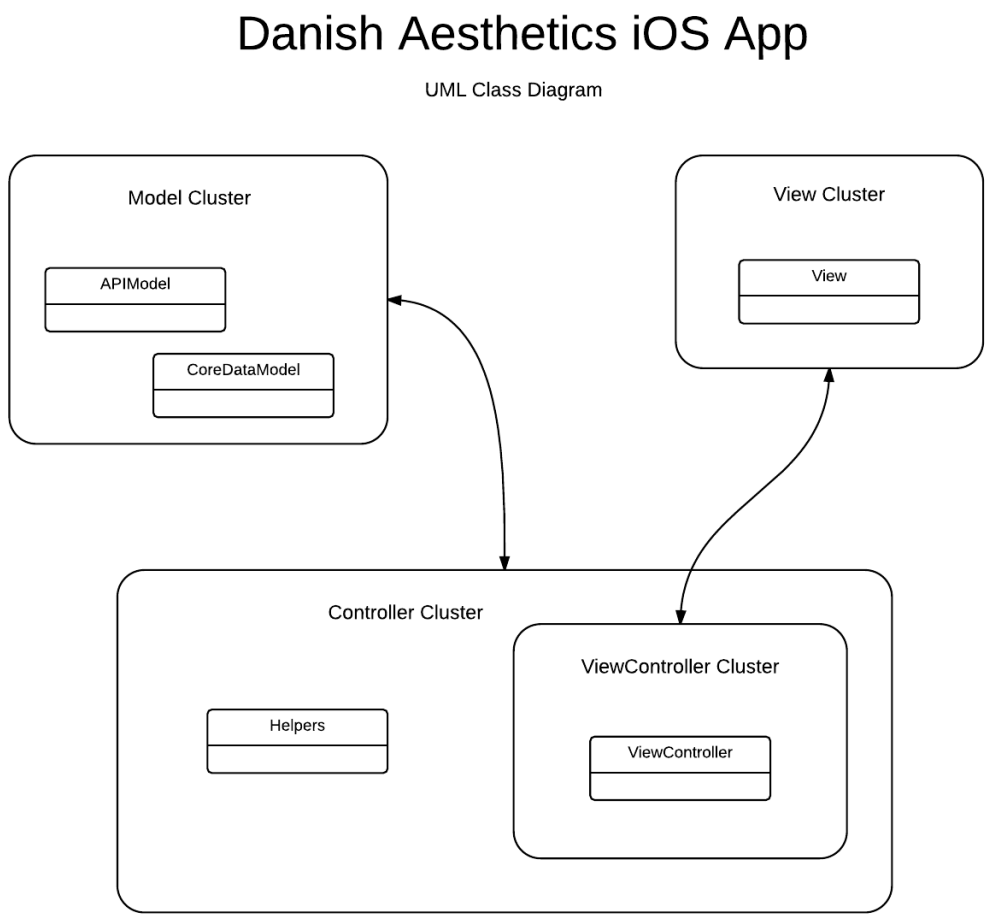


Figure 8 - UML Class Diagram

3.4 Core Data

For storing data within the iPhone, this application will utilize iOS' CoreData Framework. Apple aptly describes the CoreData Framework in their documentation:

"*The Core Data framework provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence*."[17]

The CoreData might look and seem like a relational database, but it is not.[18] It stores object on an object graph, and does not save it until it is asked to save it. This means that one is able to create, edit and delete objects and when the application is closed, none of the changes persisted. This will not be the case for this application; as soon as a change occurs, the data on the object graph will be saved in the context it was created. The CoreData Framework is used for storing all the model data on the iPhone, so the application is fully functioning without connection to the API. The models will then later be mapped to the API, but more on that later. Apple provides a very good way of designing one's CoreData models, via a graphical user interface. The source files can then be automatically generated. The class diagram for this CoreData model looks can be viewed in the figure below:

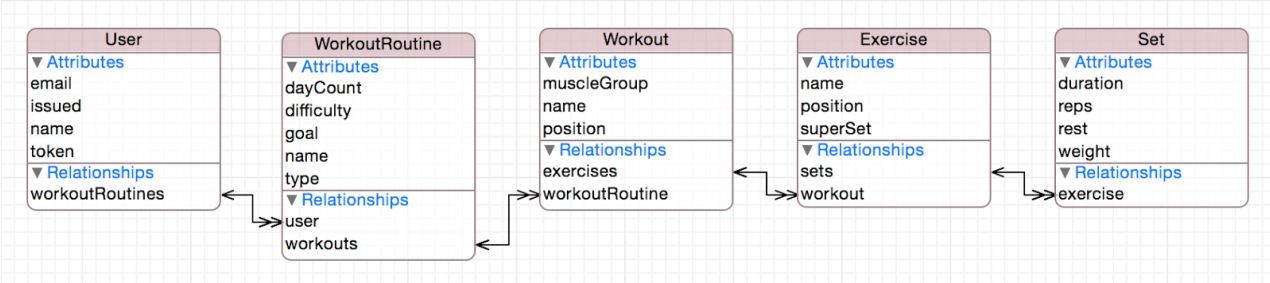


Figure 9 - CoreData Model Class Diagram

As shown in the figure above, the framework does support relationships. All relationships here are one-to-many and cascading downwards. It is not possible to delete upwards; i.e. a set cannot create its exercise. A generated source file can be viewed in the code snippet below:

```
import Foundation
import CoreData

public class User: NSManagedObject {

   @NSManaged public var email: String
   @NSManaged public var issued: String
   @NSManaged public var name: String
   @NSManaged public var token: String
   @NSManaged var workoutRoutines: NSOrderedSet

   public class func createInManagedContext(context: NSManagedObjectContext,
   Name: String, Email: String, Issued: String, Token: String) -> User {
       let newUser = NSEntityDescription
       .insertNewObjectForEntityForName("User",
       inManagedObjectContext: context) as User

       newUser.name = Name
       newUser.email = Email
       newUser.token = Token
       newUser.issued = Issued

       return newUser
   }
}
```

Code Snippet 3 - User.swift CoreData Sourcefile

The method “createInManagedContext()” is not automatically generated, this is used for creating a User in the current context like so:

```
var user = User.createInManagedContext(self.managedObjectContext, Name:
"Janis Bruno", Email: "bruno@gmail.com", Issued: "1234", Token: "dasjhjafhs-
TEST")
```

Code Snippet 4 - Creating a User on the CoreData Object Graph

The user variable now represents the user on the CoreData object graph and updating it’s attributes will change the actual user on the object graph. However, the data will not persist, when the application is terminated, since the context has not been saved. Persistence can be achieved like so:

```
func saveCoreData() {
       var error : NSError?
       if(managedObjectContext!.save(&error) ) {
           println(error?.localizedDescription)
       }
}
```

Code Snippet 5 - Saving CoreData Context

The data is now saved, and will be accessible next time the application is launched; it will even persist through shutdowns of the iPhone. The saved data can then be fetched and used again, this is achieved like so:

```
var users = [User]()

       let fetchRequest = NSFetchRequest(entityName: "User")

       if let fetchResults = managedObjectContext!
       .executeFetchRequest(fetchRequest, error: nil) as? [User] {
           users = fetchResults
       } else {
           println("no user :/")
}
```

Code Snippet 6 - Fetching CoreData

The users array now contain all User objects stored in the CoreData context. There will only ever be one User object, since all CoreData is wiped on logout. This deletion of all data is very simple, seeing that all relationships are cascading downwards. This means that deleting a User, will delete everything from the CoreData context. This deletion is achieved like so:

```
// Find the user object to to delete
let userToDelete = self.users[0]

// Delete it from the managedObjectContext
managedObjectContext?.deleteObject(userToDelete)
self.saveCoreData()
```

Code Snippet 7 - Deletion of CoreData

WorkoutsRoutine, Workout, Exercise and Set objects can be deleted in the same way. Since Set is the only model that does not have children, it is also the only object that can be deleted without any other data being deleted.

3.5 Graphical User Interface

The graphical user interface is a very important aspect of this project, since one of the main requirements is a very user friendly application. The graphical user interface will consist of one main element very common to iOS applications; the Tab Bar. This type of application will have several icons at the bottom of the screen to navigate between each tab of the application. This is illustrated in the illustration just below this section.

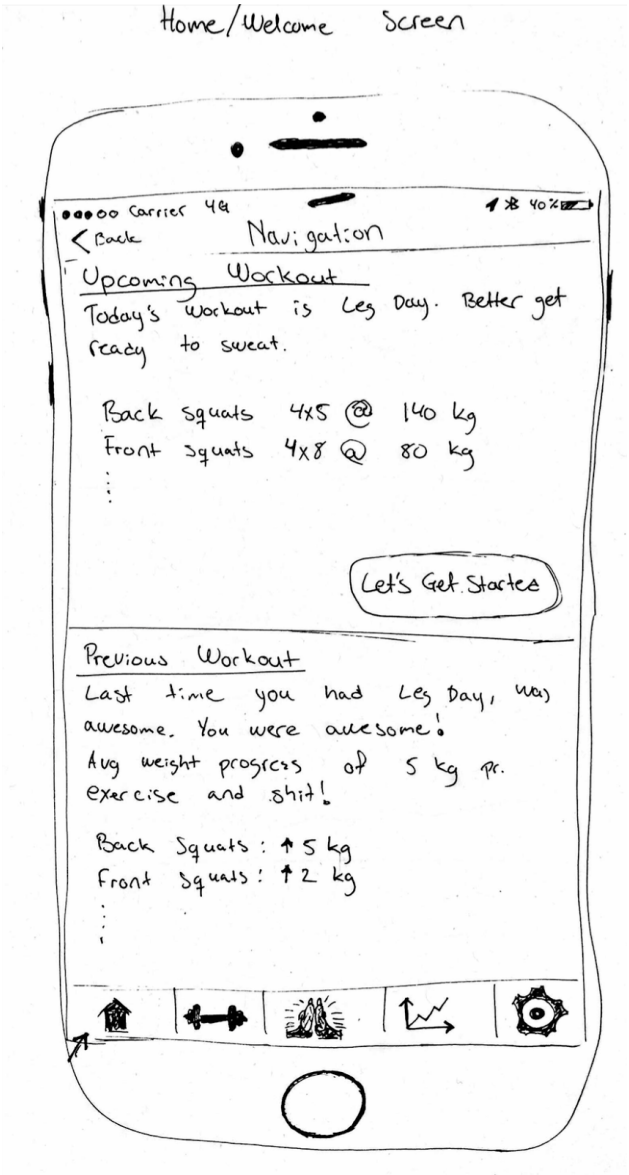


Figure 10 - Initial Mock Up

Basing an application on the tab bar has many pros, but of course also cons. One major con, is the some of the screen real estate, will always be unavailable, because it is reserved for the tab bar. This can be circumvented by hiding the tab bar in certain scenarios. A major pro, is the ability to always navigate to a different part of the application, with just one tap. This gives great freedom, instead of having to press back multiple times, before being able to navigate freely within the application. Another pro, is the fact that leaving a tab, does not mean any data is lost. For example; the user is creating a workout, and all of a sudden they cannot remember some of their statistics. The user now leaves the process of creating a workout by tapping the statistics icon in the tab bar. After the user is done reading up on their progress, they return to the workout tab, and it pops up, just like they left it. All of this is taken care of automatically and no extra programming is needed.

Besides the Tab Bar, this application will also utilize another frequently used component in iOS development; namely the Navigation Bar. The navigation bar can be seen in the illustration from before, at the top of the screen. The way a navigation bar works, is that every screen that is shown

to the user, is pushed to the navigation stack. Whenever a user taps "< back" or swipes from the left edge of the screen, the top most view is popped from the stack. The navigation bar also gives a great user experience, since the user always know exactly where they are in the application and how to get back.

A major pro for both of these components, is the fact that they are used in almost every single stock application on the iPhone; the phone application, the clock application, the settings application and many more. This also contributes to fulfill the user friendly requirement set by Danish Aesthetics, because users will instantly know how to navigate throughout the entire application the very second they open the application for the first time. All initial mockups of the application can be found in Appendix C - Graphical User Interface Mockups.

3.6 Storyboards and Views

In iOS development, views are normally not created the way that they are in other programming languages. Views are represented as part of a Storyboard. Apple perfectly describes what a storyboard is:

"*A storyboard is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens. A storyboard is composed of a sequence of scenes, each of which represents a view controller and its views; scenes are connected by segue objects, which represent a transition between two view controllers.*"[9]

This is how each screen/view of the graphical user interface is created in Xcode; first by dragging in a view controller to the storyboard and then afterwards filling it with different components. Part of the storyboard for this project can be seen in the illustration below.

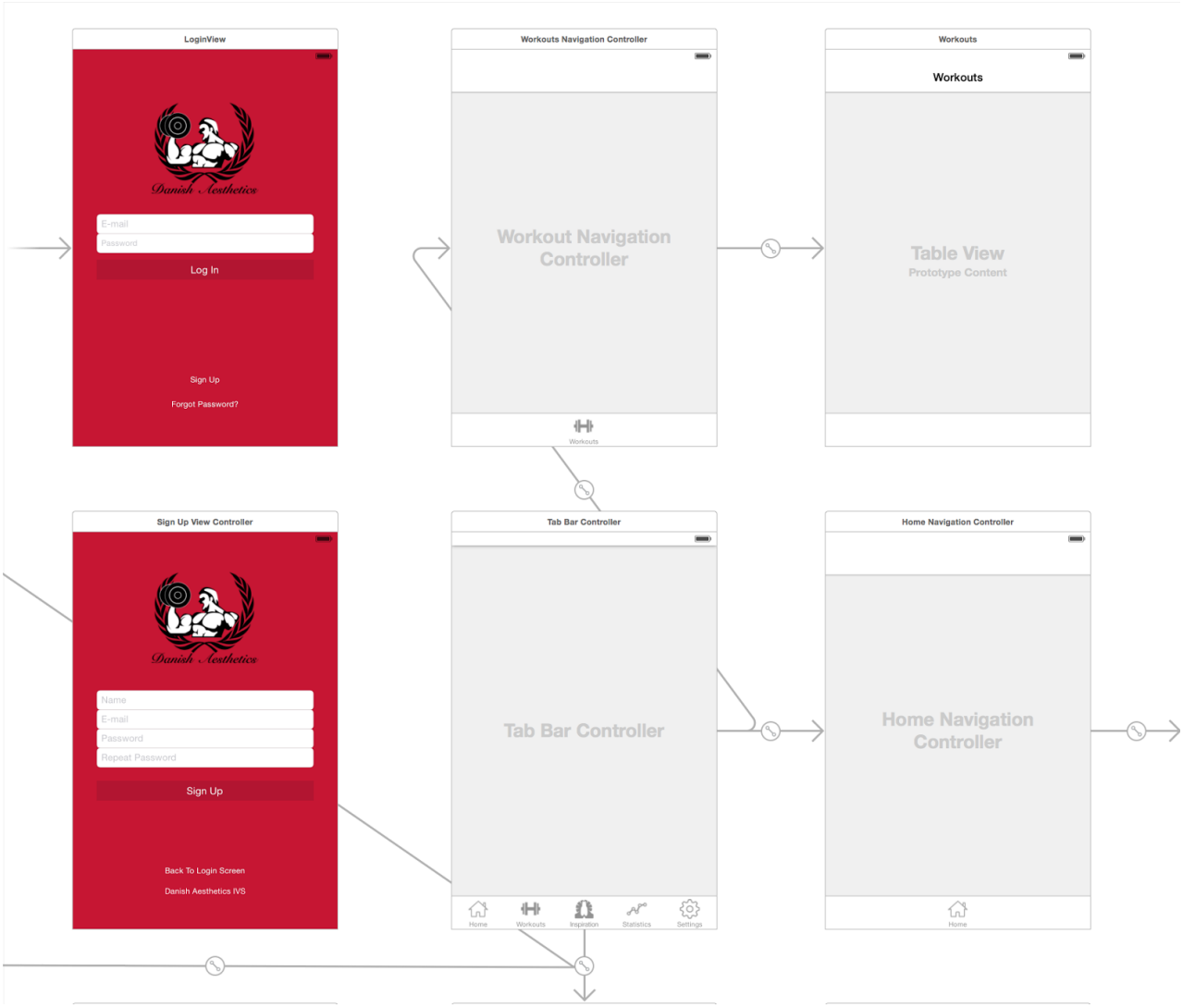


Figure 11 - Partial Storyboard

As one might be able to deduce from the above illustration, the view controller in the middle (the one with 5 icons at the bottom of the view) is the main controller for the application. It is the tab bar controller from the previous section. This controller manages the main navigation between the different tabs of the application. Each tab has it own "main controller" which in every single case is the navigation controller also mentioned in the previous section. It might be possible to discern the different navigation controllers; "Home", "Settings", "Workout", "Inspiration" and "Statistics". The topmost red screen is the starting point of the application, whenever it is launched. This is indicated by the small grey arrow. Along with the other two red screens, these constitutes the screens for signing in, creating a new user and resetting a user's password (The screen for resetting a user's password did not fit in the final screenshot. A full overview of the storyboard can be found as an appendix).

The "Workout" view, for creating workouts, and all views that are created within this navigation view, will rely heavily on iOS' UITableView object. An object which allows for displaying data in a customizable table view as the name suggests. This allows for easy implementation of features such as reloading the table when a new element is added, deletion of element via left swipes and

selection of elements at a certain index. An example of the aforementioned methods can be viewed in the code snippets below:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {

       var cell: WorkoutTableViewCell = self.workoutTableView
       .dequeueReusableCellWithIdentifier("workoutCell")
       as WorkoutTableViewCell

       if(indexPath.row == 0) {
           cell.loadItem("Add New Workout", image: "DA Logo")
       } else {
           cell.loadItem(self.wors[indexPath.row - 1].name,
           image: "DA Logo", routine: self.wors[indexPath.row - 1])
       }

       return cell

}
```

Code Snippet 8 - Loading Cells in UITableView

```
func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {

       workoutTableView.deselectRowAtIndexPath(indexPath, animated: true)
       //Do stuff !!

       var row:Int = indexPath.row
       var cell:WorkoutTableViewCell = workoutTableView
       .cellForRowAtIndexPath(indexPath) as WorkoutTableViewCell

       NSLog("Workout Row Clicked: %d", row)

       let createView = self.storyboard?
       .instantiateViewControllerWithIdentifier("CreateWorkoutView")
       as CreateWorkoutViewController

       if(row == 0) {
           //Add New Workout
           NSLog("Add New Workout Selected")
           let popUp = UIAlertController(title: "Add New Workout",
           message: "Enter the name of your new workout.",
           preferredStyle: UIAlertControllerStyle.Alert)
           popUp.addTextFieldWithConfigurationHandler(addTextField)
           popUp.addAction(UIAlertAction(title: "Cancel",
           style: UIAlertActionStyle.Default, handler: nil))
           popUp.addAction(UIAlertAction(title: "OK",
           style: UIAlertActionStyle.Default, handler: newWor))
           //Pop up alert for user to input the name of the new day, or
           select a rest day.
           presentViewController(popUp, animated: true, completion: nil)
       } else {
           NSLog("Workout Selected: " + cell.workoutName.text!)
           //Go to workout overview screen
```

```
            //TODO: Pass along the ID of the workout or something similar
            createView.workoutRoutines = [self.wors[row-1]]
        }

        self.showViewController(createView, sender: self)

}
```

Code Snippet 9 - Selecting Cells in UITableView

```
func tableView(tableView: UITableView, commitEditingStyle editingStyle:
UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
        if(editingStyle == .Delete ) {
            // Find the LogItem object the user is trying to delete
            let worToDelete = wors[indexPath.row - 1]

            // Delete it from the managedObjectContext
            managedObjectContext?.deleteObject(worToDelete)

            // Refresh the table view to indicate that it's deleted
            self.fetchWorkoutRoutines()

            // Tell the table view to animate out that row
            [workoutTableView.deleteRowsAtIndexPaths([indexPath],
            withRowAnimation: .Automatic)]

            self.saveCoreData()
        }
}
```

Code Snippet 10 - Deleting Cells in UITableView

The “Inspiration” view, for displaying workouts from Danish Aesthetics’ website, will rely on iOS’ UIWebView object. This object allows for any webpage to be loaded directly in the application. This way, all content from the website is easily displayed in a way the conforms with Danish Aesthetics’ current brand and image. The web view is created like so:

```
@IBOutlet weak var webView: UIWebView!

var url = NSURL(string: "https://danish-aesthetics.dk/workout/")

override func viewDidLoad() {
        super.viewDidLoad()

        self.websiteTitle.setTitleTextAttributes([NSFontAttributeName:
        UIFont(name: "SnellRoundhand-Black", size: 18)!],
        forState: UIControlState.Normal)

        self.view.addGestureRecognizer(leftScreenEdgeRecognizer)
        self.view.addGestureRecognizer(rightScreenEdgeRecognizer)

        var request = NSURLRequest(URL: url!)
        self.webView.loadRequest(request)
}
```

Code Snippet 11 - Creating UIWebView

To conform with the great user experience Apple offers its users in their own native web browser several features was added to this simple web view. First and foremost a UIToolBar object at the bottom of the screen, just above the tab bar, so the user can easily navigate back and forth, refresh the page, stop loading a page or go back to the initial website:

Code Snippet 12 - Storyboard Representation of UIToolBar

Other than the toolbar at the bottom of the screen, two UIScreenEdgePanGestureRecogniizers was added to respond to a user swiping from either the left or right edge of the screen. This is a coming navigation feature that is found in the native Safari web browser for iOS, that allows user to navigate back and forth instead of using the arrow buttons. The same navigation is also standard for navigating back in the navigation controller that this application also utilizes throughout. These pan gesture recognizers is implemented like so:

```
var leftScreenEdgeRecognizer: UIScreenEdgePanGestureRecognizer!
var rightScreenEdgeRecognizer: UIScreenEdgePanGestureRecognizer!

required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)

        /* Create the Pan Gesture Recognizer */
        leftScreenEdgeRecognizer =  UIScreenEdgePanGestureRecognizer(
        target: self, action: "handleLeftScreenEdgePan:")
        rightScreenEdgeRecognizer =  UIScreenEdgePanGestureRecognizer(
        target: self, action: "handleRightScreenEdgePan:")

        /* Detect pans from left edge to the inside of the view */
        leftScreenEdgeRecognizer.edges = .Left
        rightScreenEdgeRecognizer.edges = .Right
}

func handleLeftScreenEdgePan(sender: UIScreenEdgePanGestureRecognizer){

        if sender.state == .Ended{
            println("Left pan detected and ended!")
            webView.goBack()
        }
}
func handleRightScreenEdgePan(sender: UIScreenEdgePanGestureRecognizer){

        if sender.state == .Ended{
            println("Right pan detected and ended!")
            webView.goForward()
        }
}
```

Code Snippet 13 - Create UIScreenPanGestureRecognizers

The pan gesture recognizers are added to the view in the method viewDidLoad(), which can be found in the previous code snippet (Creating UIWebView - insert number). The finished web view looks like this:

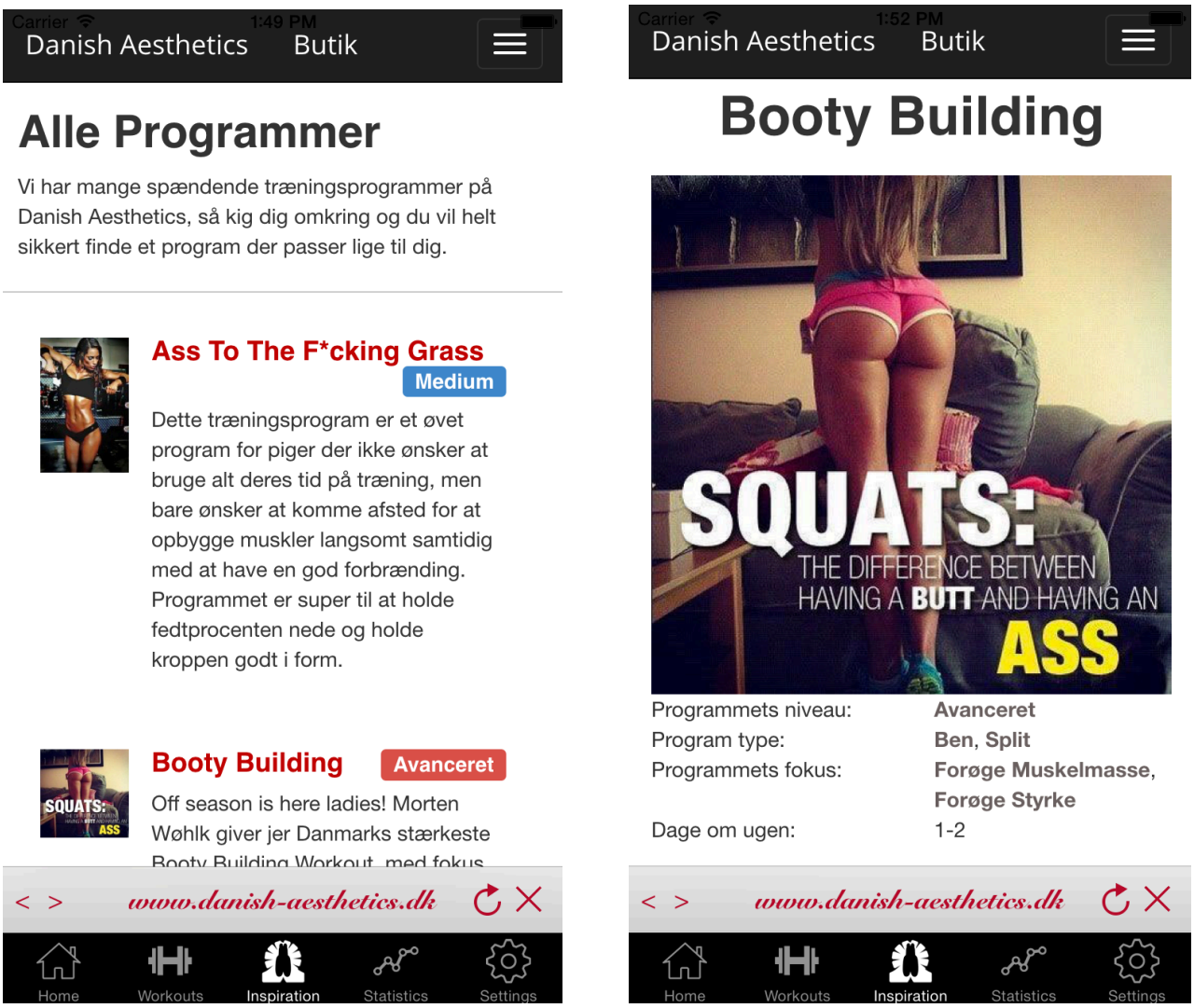


Figure 12 - WebView

3.7 Web Application Programming Interface

The application, which is built for the project, requires extensive sharing of data between the server and the client application. To provide the means of communication a set of common standards and protocols must be established between the client and the server. Since client and server sides are written in different programing languages the selection of standards and protocols are playing a crucial role and influences the architecture of the system. To make the right choice and accomplish the desired outcome the system must comply to following requirements:

- Decoupling the functionality from the presentation of the application
- Web server must be operating system independent
- Web server should not be limited to support only iOS

- Developers should have extensive knowledge of the programming language in which the web server is implemented
- Ease of manipulation and storage of the data in database
- Open Source

To fulfill the requirements a selection of frameworks, protocols and standards were considered and summarized in the following paragraph.

To satisfy the first requirement a web server must be selected. A web server is a computer system that process requests over Hypertext Transfer Protocol and the selection process is quite challenging. In almost every programming language a web server has been already developed and a skilled person can also develop one from scratch, therefore to evaluate the selection of available web servers and frameworks the criteria for the selection for the project purpose are as follow:

[5] It's efficient - fast performance and data processing
[6] Resists failures
[7] It does what the client expects and is reliable
[8] Relatively simple process of scaling and load balancing
[9] Accessibility of different vendor databases - simultaneously
[10] Only secure and trusted client access to web server

From the requirements above the selection was limited to web server applications developed in PHP or Node.js. In order to choose from two options, it has to be evaluated if the servers can run efficiently. To do that the main task is to acknowledge the complexity and performance on the data structures and algorithms which are being executed on the web server. For the project specifications the main purpose of the web server is to provide interface to the server database and respond to the basic Create, Read, Update and Delete (CRUD) functions for user data, so the data can be accessible from any device and will be synchronized across the client devices. The user data mainly consists of simple data types - strings, integers, longs, arrays, booleans and date times, the data is sanitized, validated and parsed to the MySql database. One more crucial aspect of evaluating the efficiency is how many requests a web server can simultaneously service and it is neatly summarized in graph 1 and 2 where Node.js and PHP running on Apache2 HTTP server are compared. The blue line represents response time, red line is server CPU usage and green line shows how many simultaneous user requests are being serviced.

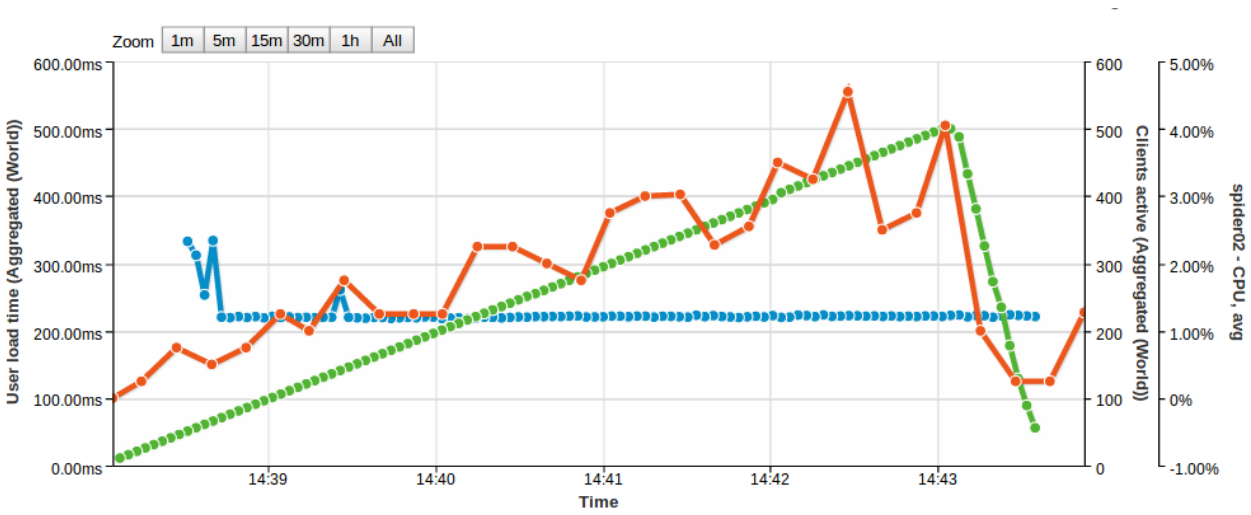


Figure 13 - NodeJS Performance

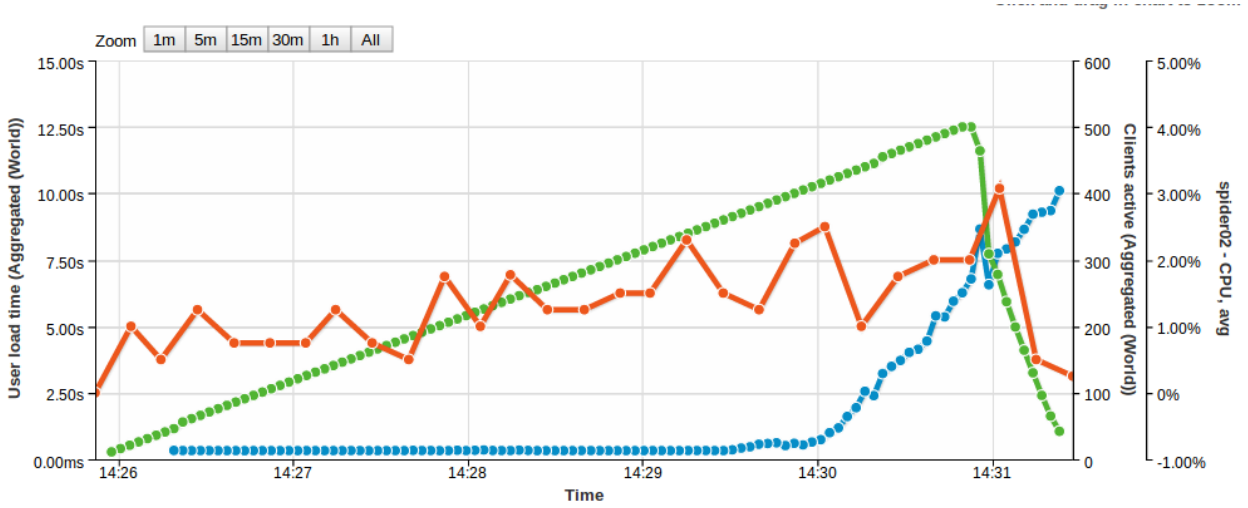


Figure 14 - PHP Apache Performance

The 2nd graph clearly shows the difference of how the response time for PHP running on Apache 2 is rapidly increasing at 340 concurrent requests but for Node.js the response time stays stable independently of concurrent requests. As the expected user base for the application is around 400 - 500 users Node.js would be the optimal choice for the project purposes as the web server. The reason why Node.js is capable of servicing many concurrent requests is well summarized by its developers:

"*Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*"[11]

In the heart of Node.js is an asynchronous event driven JavaScript framework where no function is directly performing I/O operations, therefore there are no worries of deadlocking processes and many connections can be handled concurrently, thus developers can easily develop scalable solutions. To go even one step further and bring structure to the Node.js web server a framework called Sails.js can be utilized which emulates real time Model View Controller (MVC) design pattern for Node.js and adds a bucket of useful features[12]. Sails.js allows developers to utilize a powerful data access layer with Object Relation Mapping (ORM) called Waterline which works with any database. The importance of the data access layers is the possibility to work with MySql and MongoDB simultaneously e.g. to keep high response time user data can be retrieved from MySql database while pictures are retrieved with simultaneous asynchronous request from MongoDB. To implement such a feature the only requirement is to define database adapter for each model so Waterline can recognize it e.g. User has one - to -many relationship with Picture and the values will be returned from mongo database for Picture and User from MySql database :

```
module.exports = {
   adapter: 'mysql',
   attributes: {
       name: 'string',
       email: 'string',
       picturePath : {
           collection: 'user'
```

```
            via: 'picture'
        }
    }
};
```

Code Snippet 14 - User.js Model

```
module.exports = {
    adapter: 'mongo',
    attributes: {
        path: 'string',
        name: 'string',
        user: {
            model: 'user'
        }
    }
};
```

Code Snippet 15 - Picture.js Model

Waterline achieves the access of data from different data stores by having uniform data entities and API which gives a set of functions to perform any data manipulations. The database structure itself can be created from the data models. Like from the model examples above, when the web server is started, Waterline can migrate the entities to any data store and create the tables e.g. for the User model Waterline will create a table in MySql database called users with columns: name, email and picturePath, with the data types respectively : string, string and picturePath will be dynamically populated on demand with data from Picture table. If tables already exist in the database and contain data that should not be overridden or deleted Waterline gives few options to specify what actions to be taken when the server is started. All this functionality is coming from the Waterline database adapter and if custom queries are necessary, they can be added to the model, to extend the default set of functions.

As mentioned in the overall requirements the web server should enable to decouple the functionality from the presentation, therefore data objects must be sent over from client to web server and vice versa. The target device for the project is iPhone it has access to network over Wifi or 3G/4G and therefore it can communicate with the web server, but it should not only be limited to iOS thus the system architecture should provide the communication with other platforms over common protocol. Some options to choose from that are versatile and comply with the HTTP protocol are Simple Object Access Protocol (SOAP) which is a protocol specification for exchanging structured information in the implementation of web service. SOAP internally uses Extensible Markup Language (XML) and works with messages which are structured in the following way:

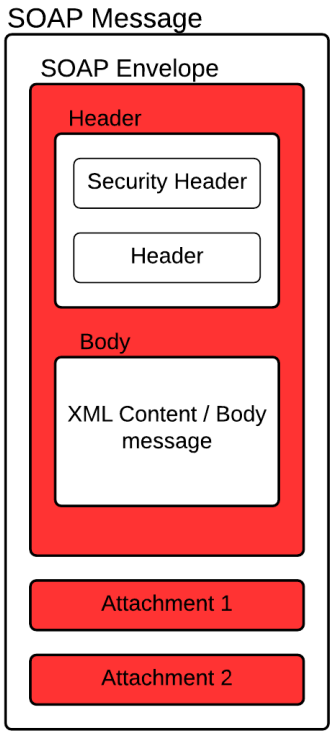


Figure 15 - SOAP Message

Request:

POST /InStock HTTP/1.1

```
Host: www.danish-aesthetics.dk
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">
<soap:Body                    xmlns:m="http://www.danish-
aesthetics.dk/user/login">
 <m:DoLogin>
  <m:UserName>janis@gmail.com</m:UserName>
  <m:Password>Izth3b3st</m:Password>
 </m:DoLogin>
</soap:Body>
</soap:Envelope>
```

Response:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">
<soap:Body                    xmlns:m="http://www.danish-
aesthetics.dk/user/login">
 <m:LoginResponse>
<m:Token>sadAS2sadWeadasd123x32.asasd.adsgndh3HD3</m:Tok
en>
 </m:LoginResponse>
</soap:Body>
</soap:Envelope>
```

Code Snippet 16 - SOAP Request & Response

As the example shows the structure of the SOAP message can be altered and does not require for all parts of the message to be included and the message itself is strictly documented and therefore the receiver should follow the structure of the message to access each part.
The other option is JavaScript Object Notation (JSON) which is an open standard format that uses human-readable text to transmit data objects consisting of key- value pairs[13]. JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values - array, vector, list or sequence

To transmit data the JSON is included in the request or response body or can be sent over as additional arguments in the request URL e.g.

Request:	Response:

http://www.localhost:1337/user/create?username=janis@gmail.com&password=IzTh3b3st&name=Janis	{ "issued": "2014-12-01T14:31:25+00:00", "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MiwiaXNzdWVkIjoiMjAxNC0xMi0wMVQxNDozMToyNSswMDowMCJ9.oVNkm-OMVk2wdrIGvCmGFBLUi1infjAFsBAdMburW1k" }

Code Snippet 17 - JSON Request & Response

JSON text format is completely language independent and is not standardized. The only requirement for JSON is that it should follow the syntax design rules described in ECMA Standard ECMA-404 [14]. The main benefit of using JSON is the ease of serialization and deserialization of the JSON content. The iPhone application is utilizing MVC design pattern and therefore the JSON data must be parsed to the model. This parsing is achieved with a simple key value lookup mechanism on the client side and can be implemented as follows:

Server response:	JSON parser in swift

[{ "workouts": [], "user": 1, "type": "split", "difficulty": "easy", "goal": "low", "name": "second", "dayCount": 2, "id": 2, }]	// takes raw response data and returns array of models
	func parseJSONIntoRoutines(json : NSMutableArray!) -> [WorkoutRoutineAPIModel] {
	var routines = [WorkoutRoutineAPIModel]()
	let js = JSON(json)
	for (index: String, subJson: JSON) in js {
	//initializin model
	var routine = WorkoutRoutineAPIModel()
	routine.name = subJson["name"].stringValue
	routine.goal = subJson["goal"].stringValue
	routine.Difficulty = subJson["difficulty"].stringValue
	routine.DayCount = subJson["dayCount"].int!
	routines.append(routine)
	}
	return routines
	}

Code Snippet 18 - JSON Parser

In the code snippet above its clearly visible that there are no complex algorithms to parse the JSON response sent from Server to swift model, which can be further utilized in the application and vice versa.

As JSON is transmitted as plain text the client data can be relatively easily be exposed on internet and as one of the requirements state that “the web server should provide secure and trusted client

access" security must be included in the consideration for the choice of standards. Even the HTTP connection can be secured to some extent with the help of Transport Layer Security protocol the messages itself must be encrypted or other mechanism must be implemented to provide additional security.
Some developers might argue that SOAP is a more mature standard for client - server data transmission and more secure because it can include in the message header a public key for encryption so the whole message can be encrypted and decrypted on demand. The header of a SOAP message is a great attribute to have, but it order to utilize it a SOAP message must be created and for every message the header added. To make the implementation more abstract and keep the security high another approach can be considered - to implement JSON Web Token (JWT). JWT is a self-contained representation of means for communication between two parties and once the token is created it can be integrated in the information flow as it suits the best for project needs. For the project JWT could be implemented so that it contains all necessary information to provide secure access and some additional information in three parts:

- header
- payload
- signature

The header could contain two things: the type of the token and the algorithm name e.g. { type : 'JWT', alg: 'SHA256' } . With those two values any client should be able to decode the token by using SHA256 decryption algorithm and extract the payload. The payload contains the information about the user which is necessary to restrict or allow specific user or user groups to access specific parts of the API. The payload could be implemented as follows : { userid: 1, admin: true, email:janis@gmail.com } and decoding the payload the server can determine the identity of the user without exposing his password. For the signature the header and the payload are used and encoded into base64, an additional secret key is added by the web server and the the whole data structure is signed with the algorithm specified in the header.

The output of the generated token should provide a result like : *xxxxxxx.yyyy.zzzzzzz* . Where the *x* is the encoded header, the *y* is the encoded payload and the z is the signature and for convenience a dot is used to separate the parts. The server or the application can decode the *yyyy* part which contains the payload and identify the user if necessary not only by the token but by the user credentials or unique identifier.

In order to get the token the user must register on the server and by obtaining the token the user can access the web server and the most crucial part is that the token is never stored on the server neither in database nor in cached values or cookies and that complies with the RESTful standards[15]. To identify the user the web server validates the token and from the token extracts the user id to compare against the database and return data if the user exists otherwise the web server denies the permission to access it. The flow of fully implemented JWT authentications would work as follows:

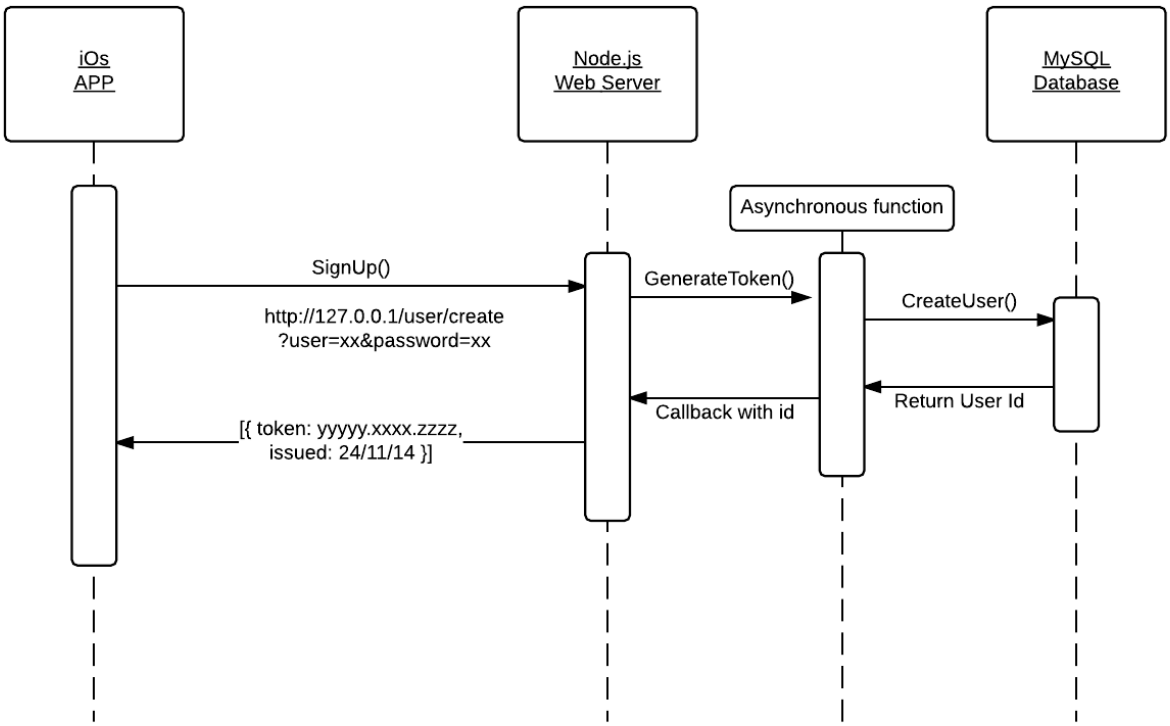


Figure 16 - JWT Authentication Sequence Diagram

When the user signs up the server takes the username and password, adds new *User* in the database, returns the *ID* of newly created user and generates a new token which is sent back to the user signed with the SHA256 encryption algorithm. Once the token is issued and sent back to the user it must stored on the client side and included in each HTTP requests header so when a user makes any request the server side can validate it and based on the control flow perform the desired action. For example if user wants to get his profile details the sequence flow for *getUserDetails()* function would be as follows[]:

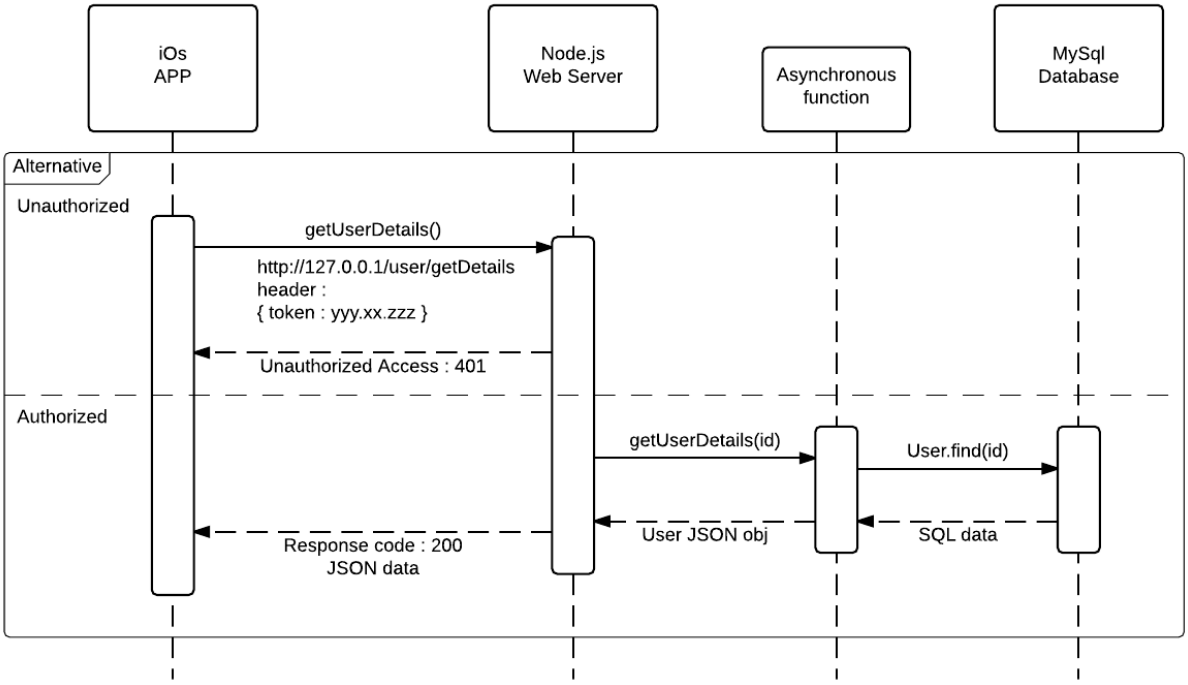


Figure 17 - Unathorized vs. Authorized Sequence Diagram

Even if someone tries to decode the token and change the payload to grant more permissions than designated, the token will not be valid and the authentication will be rejected because the signature of the token will be compromised. The only chance would be for a someone to obtain the security key from the server and encrypt the token, but that can be prevented with additional encryptions of the key. If something goes wrong on the client side with the token, the client will be notified and they can obtain a new token just by signing in the application again with their credentials.

From the design choices described above the best solution is to implement the web server with Sails.js which allows to integrate all selected standards, frameworks and protocols mentioned above. Another benefit is that Sails.js is fully written in JavaScript and as both developers have extensive experience in web page development with JavaScript so the context switching between programming languages is really small and allows to efficiently develop features for the web server. Sails.js on initial creation adds a good structure to the web server project. As mentioned before Sails.js main task is to add real time MVC pattern to Node.js thus the project solution will looks as in [Ref. to picture]. As shown in Domain Model each model is added in the project solution under the Model folder and default CRUD functions are added into separate Controllers under the Controller folder e.g. User.js model will have corresponding UserController.js which will provide the default CRUD actions and any custom action what is necessary to fulfill project requirements. The URL route mapping between the HTTP requests and

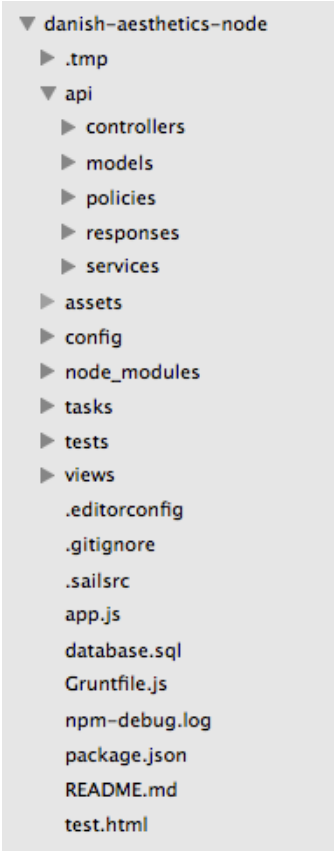


Figure 18 - Web Server Structure

controller actions are done through the routes.js located in the config folder and it allows to direct the request to a particular controller or view which will be displayed. When a user logs in from the application a request like http://127.0.0.1:1337/user/login is created. Then Sails.js first of all looks through all routes and checks if such a route is defined. If the route exists the request is handled. To facilitate the JWT authentication an authentication policy can be added under the policies folder and before the actions are handled, by simply specifying in the policies.js, under config folder, which policies must be performed for each controller, before the controller action is executed, so the authentication can take place.

Furthermore developers can easily ensure that the web server is reliable by adding automated server failure prevention scripts which will automatically restart the server in case it goes down from unexpected reasons. To add the failure prevention a common Command Line Interface (CLI) called Forever[16] can be installed on the server and the Sails application started with following command:

-prod start app.js -x -- --prod

Forever CLI will start to watch the server and in case it crashes, the server will be automatically restarted, thus ensuring that the service will always be online and users can always expect it to be running.

4. Implementation

4.1 iOS Application

The implementation of the application is very simple; simply submit the finished application to Apple for review. If the application passes Apple's review, it will be available on the Apple App Store within a couple of weeks, depending on the review time. After the application has reached the App Store the only thing left is for the user to download it to their iPhone through the store.

4.2 Web Server

First of all to run the server a host must be found where to host the application. As mentioned in the Web Server design the web server itself is self containing instance thus the only requirement from the host is to support node.js application hosting and make available access to the Node Package Modules repository, so the sails.js application can download any required dependencies on demand. Another requirement is to set up MySql database on the server which can be accessed by the web server.

Secondly the server must be copied to the directory where which has access privileges from the web users and the web server must be lifted by with the Forever[16] by calling

-prod start app.js -x -- --prod

If the web server application has started correctly, in terminal the administrator will be prompted with following image:

Figure 19 - Sails.js Production

5. Test

5.1 iOS Application Test

The main aspect of the application that needs testing, is whether or not the use of iOS' CoreData functionality works as expected. In order to test this, Unit Tests were created. Three different test cases were created for this purpose:

- Creation of data on the CoreData Object Graph
- Deletion of data on the CoreData Object Graph
- Cascading deletion on the CoreData Object Graph

Before each test is run, the test is set up by creating a User like so:

```
override func setUp() {
        super.setUp()
        // Creating the User
        User.createInManagedContext(self.managedObjectContext!, Name: "Janis
        Bruno", Email: "bruno@gmail.com", Issued: "1234", Token:
        "dasjhjafhs-TEST")
    }
```

Code Snippet 19 - Unit Test Set Up

The first test case then tries to fetch the user from the CoreData and asserts whether the User was created with the specified values like so:

```
func testCreateUserCoreData() {

       var users = [User]()

       let fetchRequest2 = NSFetchRequest(entityName: "User")

       if let fetchResults2 = self.managedObjectContext!
       .executeFetchRequest(fetchRequest2, error: nil) as? [User] {
           users = fetchResults2
       }

       XCTAssert(users[0].name == "Janis Bruno", "Name - Pass")
       XCTAssert(users[0].email == "bruno@gmail.com", "Email - Pass")
       XCTAssert(users[0].issued == "1234", "Issued - Pass")
       XCTAssert(users[0].token == "dasjhjafhs-TEST", "Token - Pass")
   }
```

Code Snippet 20 - Create CoreData Unit Test

The next test case will fetch the User again, verify that the User is created then deleting it and verifying that there is no longer any user on the CoreData Object Graph like so:

```
func testDeleteUserCoreData() {

       var users = [User]()

       let fetchRequest = NSFetchRequest(entityName: "User")

       if let fetchResults = self.managedObjectContext!
       .executeFetchRequest(fetchRequest, error: nil) as? [User] {
           users = fetchResults
       }

       XCTAssert(users.count == 1, "User Not Deleted - Pass")

       let userToDelete = users[0]

       managedObjectContext?.deleteObject(userToDelete)

       if let fetchResults2 = self.managedObjectContext!
       .executeFetchRequest(fetchRequest, error: nil) as? [User] {
           users = fetchResults2
       }

       XCTAssert(users.count == 0, "User Deleted - Pass")

   }
```

Code Snippet 21 - Delete CoreData Unit Test

The last test case will create three WorkoutRoutines for the User created during setup, fetch all workout routines and verify three workouts were in fact created. The test then deletes the user that was initially fetched, fetches all workout routines again and verifies that no workout routines exist like so:

```
func testDeleteUserCoreDataCascade() {
       var users = [User]()
       var workouts = [WorkoutRoutine]()

       let fetchRequest = NSFetchRequest(entityName: "User")

       if let fetchResults = self.managedObjectContext!
       .executeFetchRequest(fetchRequest, error: nil) as? [User] {
           users = fetchResults
       }

       WorkoutRoutine.createInManagedContext(self.managedObjectContext!,
       Name: "Test Workout 1", User: users[0])
       WorkoutRoutine.createInManagedContext(self.managedObjectContext!,
       Name: "Test Workout 2", User: users[0])
       WorkoutRoutine.createInManagedContext(self.managedObjectContext!,
       Name: "Test Workout 3", User: users[0])

       let fetchRequest2 = NSFetchRequest(entityName: "WorkoutRoutine")

       if let fetchResults2 = self.managedObjectContext!
       .executeFetchRequest(fetchRequest2, error: nil) as? [WorkoutRoutine]
{
           workouts = fetchResults2
       }

       XCTAssert(workouts.count == 3, "Workouts Created - Pass")

       self.managedObjectContext?.deleteObject(users[0])

       let fetchRequest3 = NSFetchRequest(entityName: "WorkoutRoutine")

       if let fetchResults3 = self.managedObjectContext!
       .executeFetchRequest(fetchRequest3, error: nil) as? [WorkoutRoutine]
{
           workouts = fetchResults3
       }

       XCTAssert(workouts.count == 0, "Cascade Delete - Pass")

   }
```

Code Snippet 22 - Cascade Delete Unit Test

Since all operations with iOS' CoreData is called and implemented in the exact same way for all CoreData models work in the same way, these three test cases verifies that all operations with iOS' CoreData, used throughout the application, works as expected.

5.2 Web Server Test

A big part of fully functioning system architecture relies on data transmission between the iOS application and the Web Server, therefore the HTTP requests must be tested. The tests should prove that the implementation of the web server controllers are correct and reliable and they can be accessed from the iOS application.

To test the HTTP requests a tool called POSTMAN[22] is used. POSTMAN is designed to create and send custom HTTP requests to web servers and it displays the response in the user interface. The tests are performed during the implementation process and they must be performed manually to verify that the controllers are sending back the expected response. To verify if the controllers on the Web Server are working the following test cases were made:

- Create user
- Login with user
- Add user workout routine
- Get all workout routines

Before the tests are run the web server must be started. To start the web server a user with the right privileges must navigate to the server's path from the terminal, where the web server is located. Than the user must enter the following command:

sails lift

If the server has started correctly than the user will be promoted with the following output in the terminal:

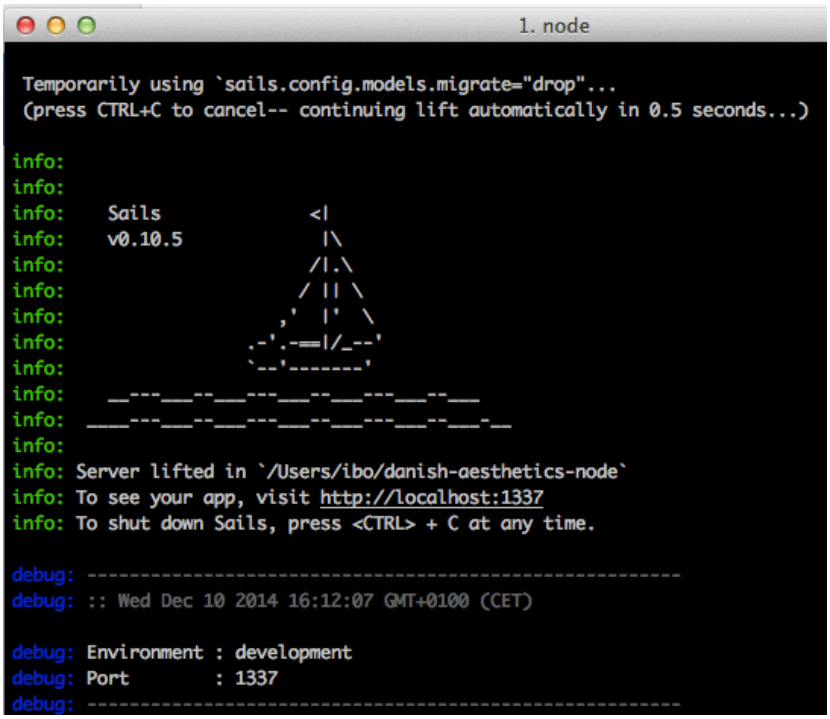


Figure 20 - Sails Terminal

Now the environment for the testing is running and from the POSTMAN client the test request can be performed. To test the "Create user" test case a request is created with the following parameters in the url: username, password and name.

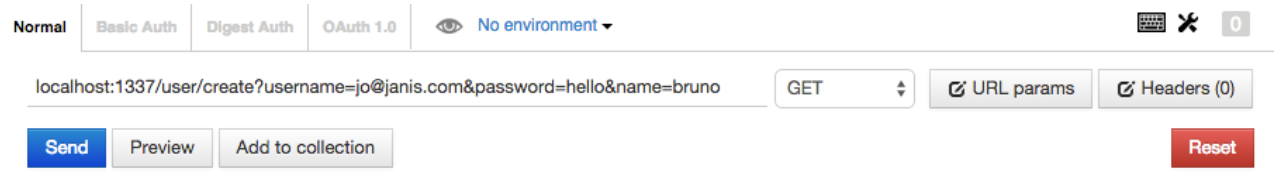


Figure 21 - POSTMAN: Create User Test

In the success scenario the request must return an HTTP response code 201 and the authentication token with the creation date. If the token and the date is returned it confirms that the JWT authentication is working and the user is saved in the database. The result should be returned as JSON object in the following format:

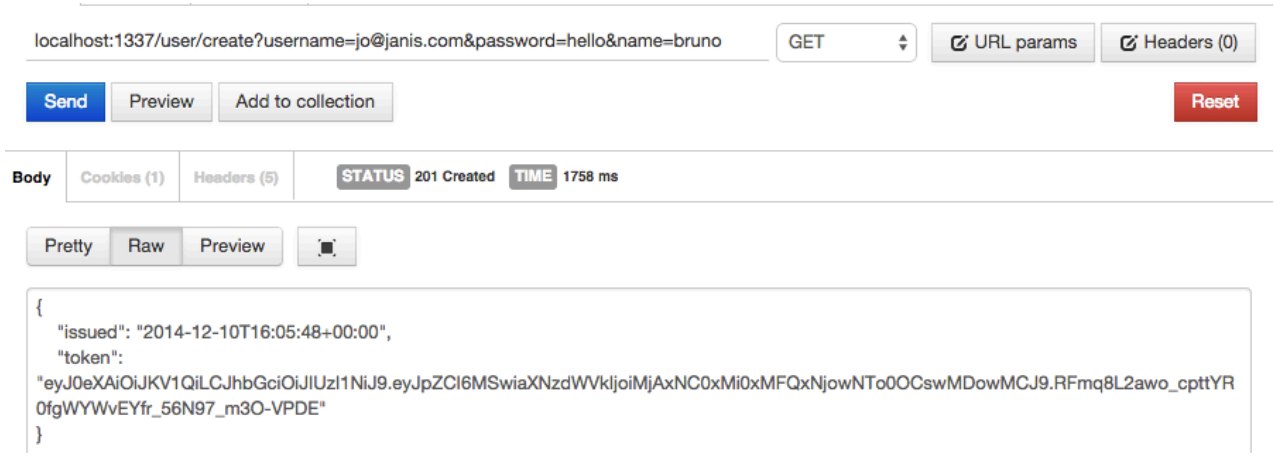


Figure 22 - POSTMAN: Create User Response

In unsuccessful scenario the HTTP response code will be 500 and in the response body a message will be shown “internal server error”.

To test the “Log in” test case the user must be created and therefore the /user/create request must must be performed first. If the “Create user” test case is successful the login request must be successful too. The HTTP request looks as follows:

Figure 23 - POSTMAN: User Login Test

In successful scenario the response from the server will have HTTP status code 200 and in the response body a JSON object will be returned with the authentication token and the date as follows:

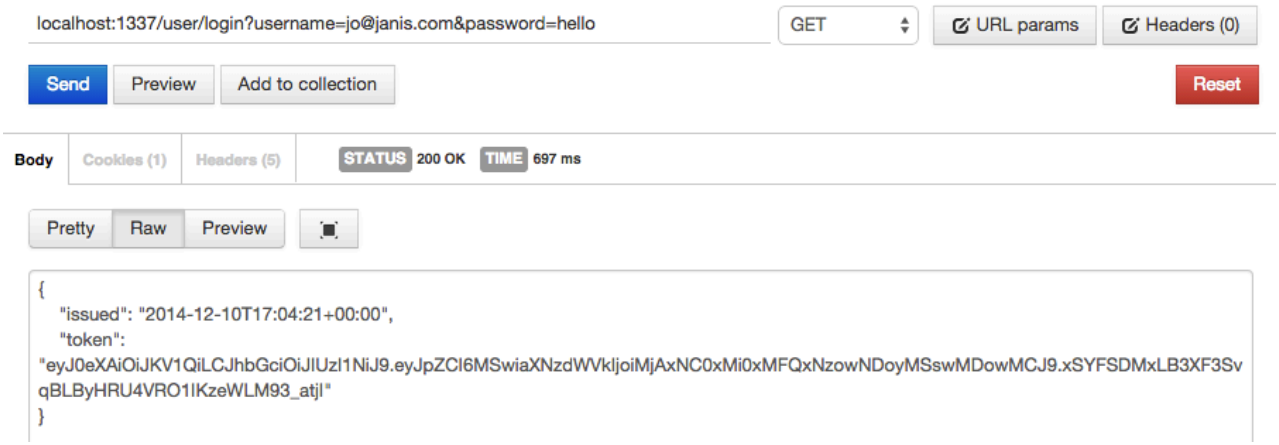


Figure 24 - POSTMAN: User Login Response

The successful scenario confirms that the user has been retrieved from the database with the correct authentication token.
In unsuccessful test scenario the response from the server will be returned with HTTP status code 400 and a message in the response body "Unauthorized".

For the "Add user workout routine" test case a POST request must be created with the following request parameters: name, position and username

Figure 25 - POSTMAN: Add Workout Routine Test

In successful scenario the response from the server will return HTTP status code 200. The status code confirms that the workout routine has been added for the particular user in the database and the relation between the user and the workout routine has been made.
Furthermore the test case "Get all workout routines" can confirm that the workout routine has been created in the database. The GET request must look like in the figure 26

Figure 26 - POSTMAN: Get All Workout Routines Test

The response from the request must return in the response body JSON object with array of all workout routines what are saved in the MySql database. In the test cases successful scenario the returned JSON object contains many workout routines and a workout routine with a name "hello" is returned, what confirms that it was added with the /adduserroutine request.

6. Discussion

The application and web API is fully functional, but due to lack of time the tracking feature will most likely not be finished by the deadline of this project. The swift programming language is very new and turned out to be harder to work with than first expected. The syntax of the swift language changed a couple of times during the project period, which caused some noticeable delays in the development. Besides the tracking feature, all requirements posed in the project description has been met; the application pulls content from the existing website and displays it in the application, the user is able to create their own workout routines and the data is accessible not only in the application but wherever Danish Aesthetics needs it, thanks to the web API.
The future of this application will be to have the tracking feature implemented, before the final examination along with the statistical presentation of the user progress in the gym. After these features have been implemented, the next step for the application, will be the possibility to actually use the workouts posted on the website as trackable workouts in the application. When these features have been implemented it is time for the application to give the users even more; Danish Aesthetics is in the process of making tutorial videos for the most common exercises and uploading them to their youtube channel. These videos should be available to users, when they are adding an exercise or when they are in the gym tracking an exercise. These features will conclude the main building stones of the application and the next step for the application will be some sort of social service within the application. It is still uncertain how this social service should work; should a user be able to share their own workouts? Should they earn experience/rank by advancing in weight on their workout routines? Should they be able to help other users by giving them advice somehow? There are tons of possibilities and different directions to go with the application and Danish Aesthetics will have to consider this carefully, before jumping into further development of the application.

7. Conclusion

The objective was to build an iPhone application, which can present the content from Danish Aesthetics website in user friendly and interactive way. This requirement was achieved to the fullest, by using their already existing responsive website directly in the application it is even done in a way that complies with Danish Aesthetics image and brand. As mentioned in the discussion, the tracking feature is, at the time of writing, not fully implemented, however the web API is. Seeing as the swift language showed to be more uphill than expected, the state of the project at the time of delivery is still very satisfactory. The scope of the project was very big, even if swift had not shown to be somewhat of an obstacle, completing all requirements before the deadline would have been a miracle. This project is a process still not finished, and that is not unsatisfactory. The team will continue to work on the application until the final presentation in January. After this, the team will split up and further development will only be done by Danish Aesthetics. All in all, a very satisfactory project and results and taking the circumstances into consideration the project is beyond satisfactory.

8. Literature

[1] JSON web token 24.09.2014 (http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-19)

[2] Efficient JSON in Swift with Functional Concepts and Generics http://robots.thoughtbot.com/efficient-json-in-swift-with-functional-concepts-and-generics - October 2014

[3] HTTP requests in Swift - https://github.com/RajeevRShephertz/HTTPRequestsInSwift/blob/master/HTTPRequestsInSwift/HTTPRequestsInSwift/DetailViewController.swift - October 2014

[4] Angular Tips - http://angular-tips.com/blog/2014/05/json-web-tokens-introduction/ - October 2014

[5] Angular Tips 2 - http://angular-tips.com/blog/2014/05/json-web-tokens-examples/ - October 2014

[6] Tokens - https://auth0.com/blog/2014/01/07/angularjs-authentication-with-cookies-vs-token/ - October 2014

[7] Tokens 2 - https://auth0.com/blog/2014/01/27/ten-things-you-should-know-about-tokens-and-cookies/ - October 2014

[8] Swift Manual - https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ - November 2014

[9] Story Boards - https://developer.apple.com/library/prerelease/ios/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html - November 2014

[10] "Node.js vs PHP - using Impact to visualize node.js efficiency" http://blog.loadimpact.com/2013/02/01/node-js-vs-php-using-load-impact-to-visualize-node-js-efficency/ - November 2014

[11] Node.js - http://nodejs.org - November 2014

[12] Sails.js features - http://sailsjs.org/#/features - November 2014

[13] JSON wikipedia - http://en.wikipedia.org/wiki/JSON - November 2014

[14] ECMA 404 Standard - http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf - December 2014

[15] REST standard - http://www.w3.org/2001/sw/wiki/REST - December 2014

[16] Forever GitHub - https://github.com/nodejitsu/forever - December 2014

[17] CoreData - https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/Articles/cdTechnologyOverview.html#//apple_ref/doc/uid/TP40009296-SW1 - December 2014

[18] CoreData 2 - https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/Articles/cdTechnologyOverview.html#//apple_ref/doc/uid/TP40009296-SW1 - December 2014

[19] Apple MVC - https://developer.apple.com/library/IOs/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html - December 2014

[20] Apple MVC 2 - https://developer.apple.com/library/IOs/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html#//apple_ref/doc/uid/TP40010810-CH14-SW14 - December 2014

[21] Apple View Controllers - https://developer.apple.com/library/ios/featuredarticles/viewcontrollerpgforiphoneos/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007457-CH1-SW1 - December 2014
[22] Postman - http://www.getpostman.com - December 2014

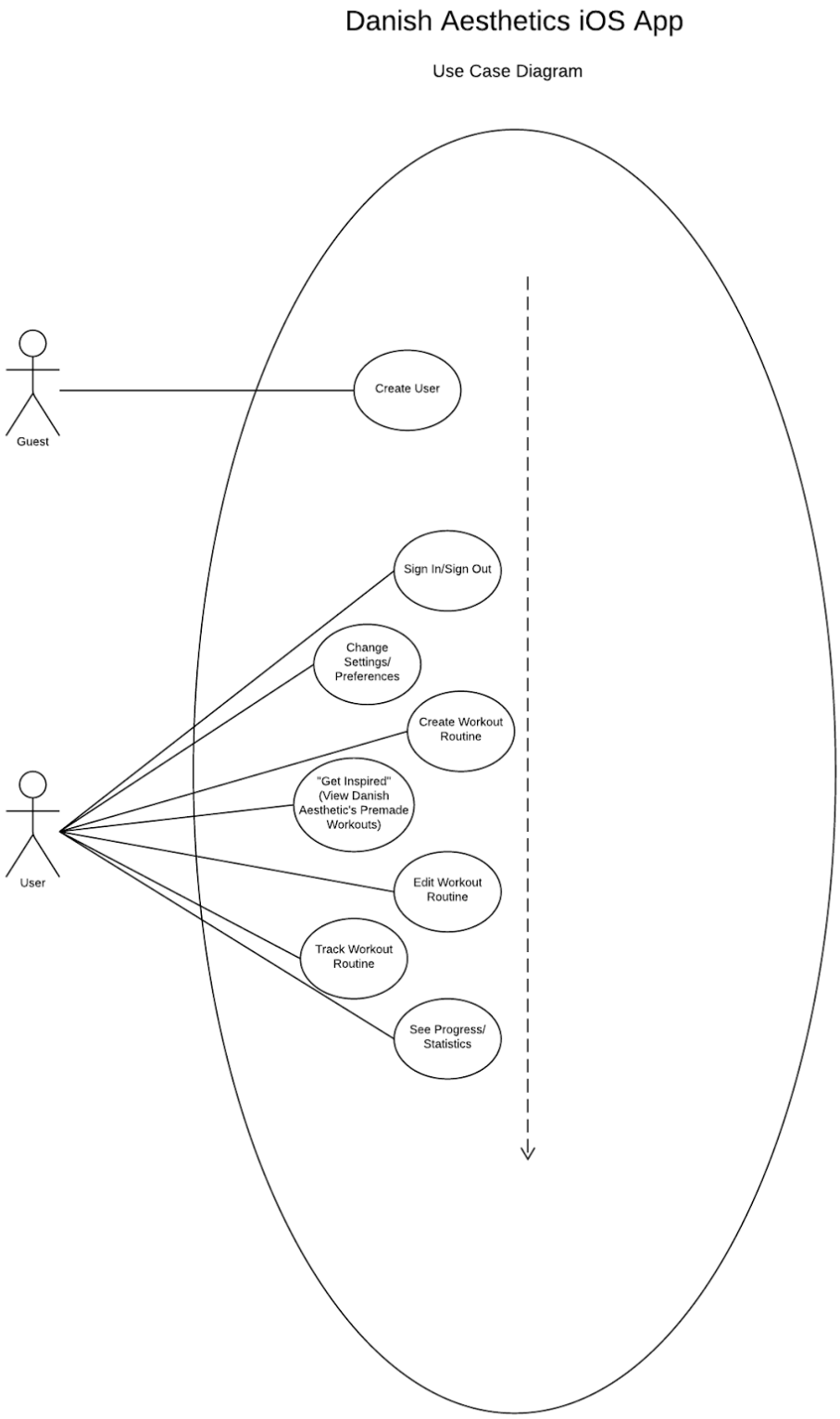
9. Appendices

9.1 Appendix A – Questionnaire

This appendix can be found in the Basic Files folder.

9.2 Appendix B – Use Cases

9.2.1 Use Case Diagram



9.2.2 Use Case Descriptions and Activity Diagrams

Create User

Use Case Name	Create User
Use Case Purpose	The purpose of this use case is to allow guest create a new user and access the application
Preconditions	
Postconditions	Guest becomes a registred system user
Limitations	
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone. Guest has Wi-Fi or 3G/4G network connection

Basic flow

1. Guest opens iPhone application
2. Gust enters name or nickname
3. Guest enters email address
4. Guest enters password and repeats the password in confirmation field
5. Guest presses “Create account” button
6. Guest gets redirected to Home screen of the application

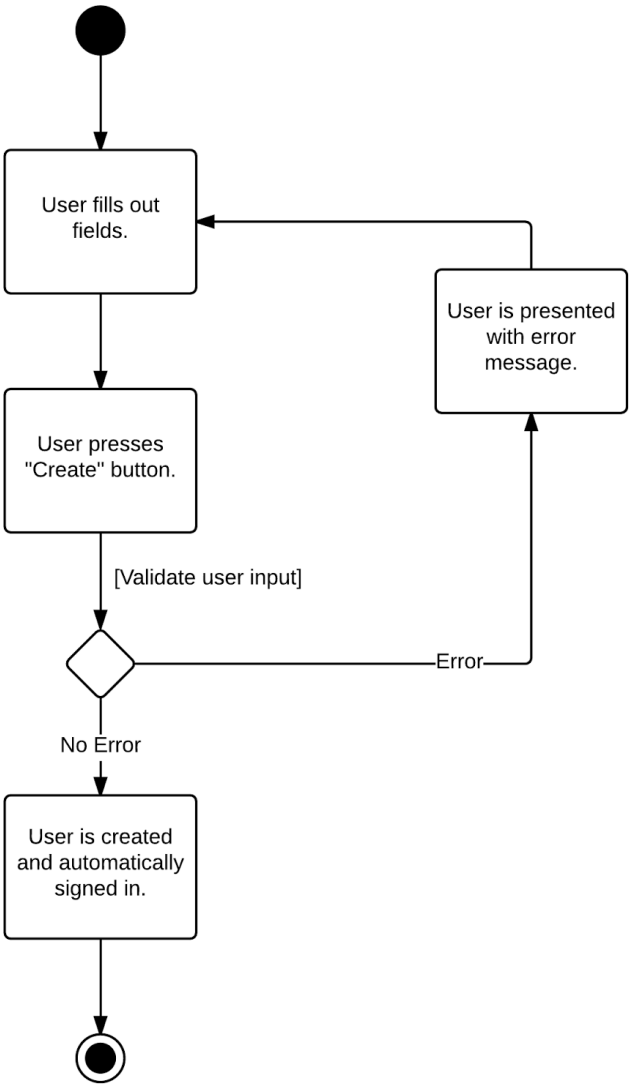
Alternative flow

Condition: Input model invalid

1. Guest performs step 1 - 5 from basic flow
2. Application alerts user with a message : “input field invalid”
3. Guest corrects input field
4. Guest presses “Create account” button
5. Guest gets redirected to Home screen of the application

Condition: user exists

1. Guest performs step 1 - 5 from basic flow
2. Application alerts with a message: “email is already registered”
3. Guest selects activity to follow from options : “Forgot password” or “Sign In”



Sign In

Use Case Name	Sign In
Use Case Purpose	The purpose of this use case is to allow a user to sign in to the application with existing user account.
Preconditions	User account exists
Postconditions	User is signed in and able to access his data
Limitations	
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone. Guest has Wi-Fi or 3G/4G network connection

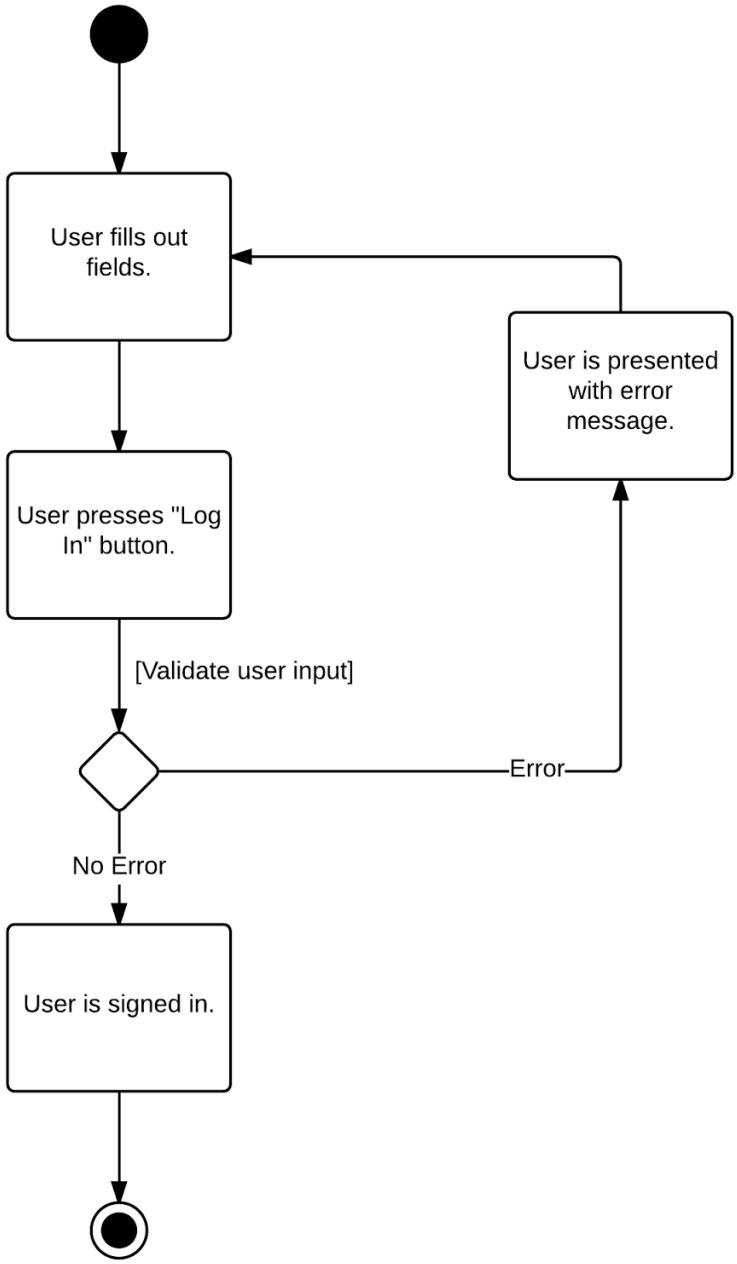
Basic flow

1. User opens the application
2. User enters email and password
3. User presses login button
4. User gets redirected to Home screen of the application

Alternative flow

Condition that triggers alternate flow: incorrect email

1. User performs step 1 - 3 from basic flow
2. Application alerts user with a message: “User with provided email does not exist”
3. User selects activity to follow: “Create User” or “Forgot Password”



Sign Out

Use Case Name	Sign Out
Use Case Purpose	The purpose of this use case is to user to logout from application and remove application data from the phone
Preconditions	Application is installed on the device. User has existing user account. User has logged in the application
Postconditions	User data is removed from device
Limitations	User data is not removed from API database
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone. Guest has Wi-Fi or 3G/4G network connection

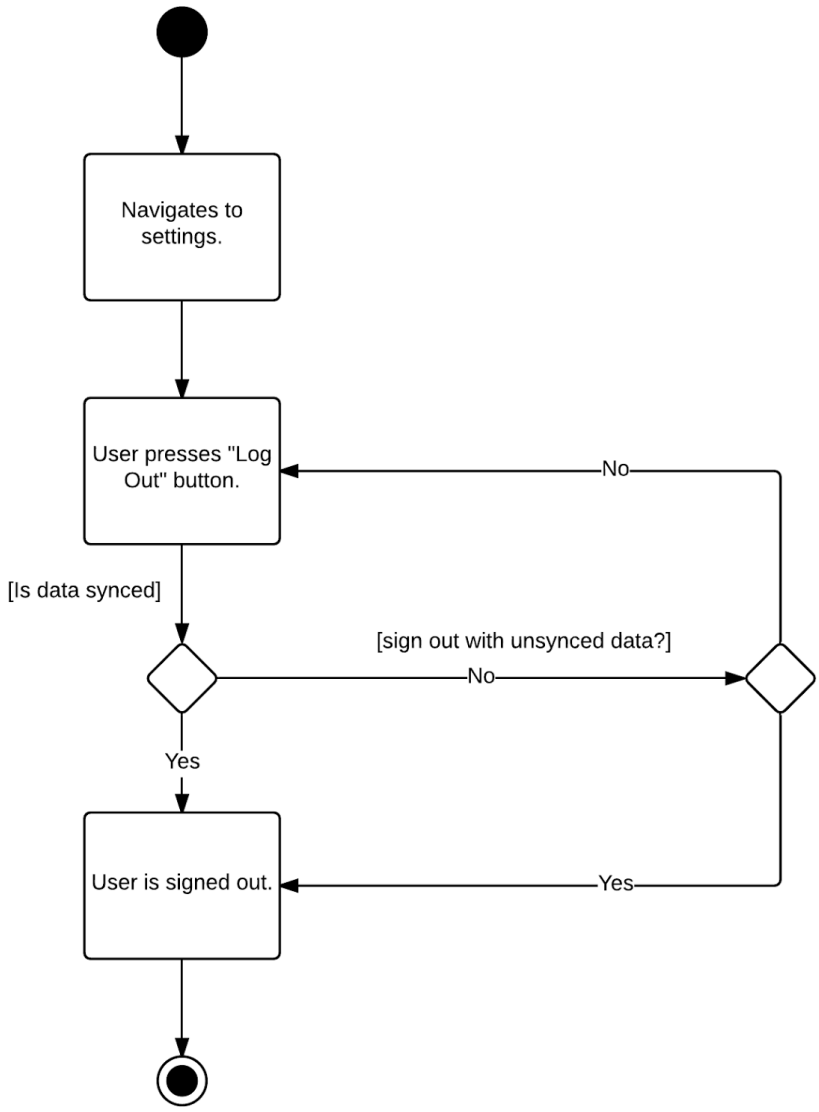
Basic flow

1. User navigates to system settings view
2. User presses button “Log Out” button
3. Application alerts user with confirmation message
4. User presses Ok and Login screen appears

Alternative flow

Condition: User refuses to log out

1. User performs steps 1 - 3 from basic flow
2. User selects “Cancel” activity
3. Application returns user to settings screen

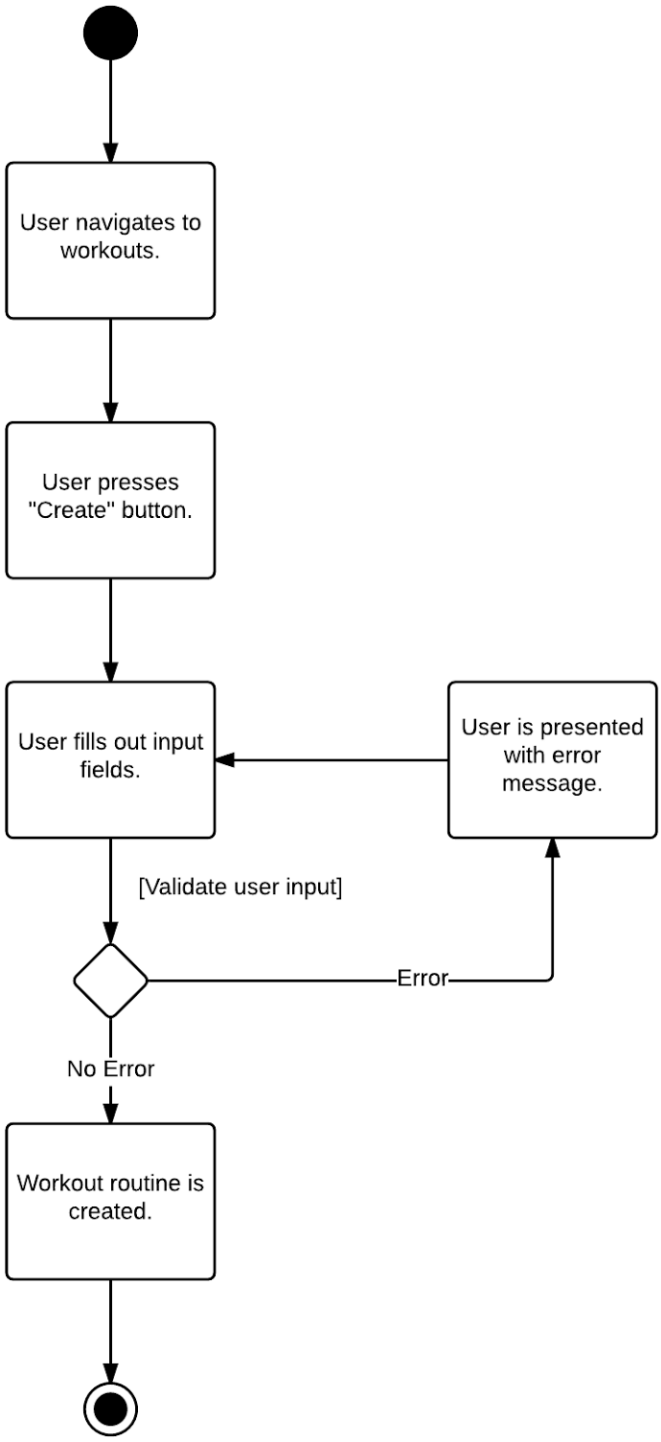


Create Workout Routine

Use Case Name	Create Workout Routine
Use Case Purpose	The purpose of this use case is to allow user to create a new workout routine
Preconditions	User has launched the application
Postconditions	Workout routine is stored on the application
Limitations	Data is not simultaneously updated in the API database
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone.

Basic flow

1. User navigates to Workout Routine screen
2. User presses “Add new Workout” button
3. User enters the name of the workout routine
4. User additionally enters difficulty and workout type
5. User presses “Done” button
6. User is navigated to “Add Workout action”

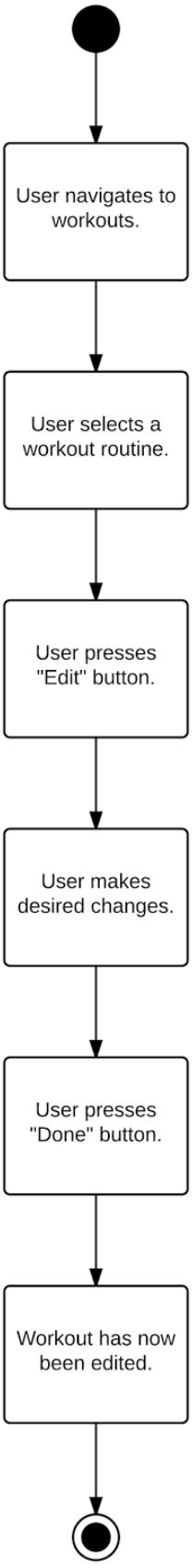


Edit Workout Routine

Use Case Name	Edit Workout Routine
Use Case Purpose	The purpose of this use case is to allow user to edit an existing workout routine
Preconditions	User has launched the application. User has existing workout routine
Postconditions	Workout routine data is updated and saved
Limitations	Data is not simultaneously updated in the API database
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone.

Basic flow

1. User navigates to Workout Routine screen
2. User selects one Workout Routine from the list of Workout Routines
3. User presses “Edit” button
4. User makes changes in the Workout Routine
5. User presses “Done” button
6. User is navigated to Workout Routine list screen


User navigates to workouts.
User selects a workout routine.
User presses "Edit" button.
User makes desired changes.
User presses "Done" button.
Workout has now been edited.

Get Inspired

Use Case Name	“Get inspired”
Use Case Purpose	The purpose of this use case is to allow application user to read articles add provided workout routines or exercises to his Workout Routine plan
Preconditions	User has launched the application. User has logged in the application.
Postconditions	User data is removed from device
Limitations	User data is not removed from API database
Assumptions	Guest has iPhone. Guest has downloaded and installed the application on his phone. Guest has Wi-Fi or 3G/4G network connection

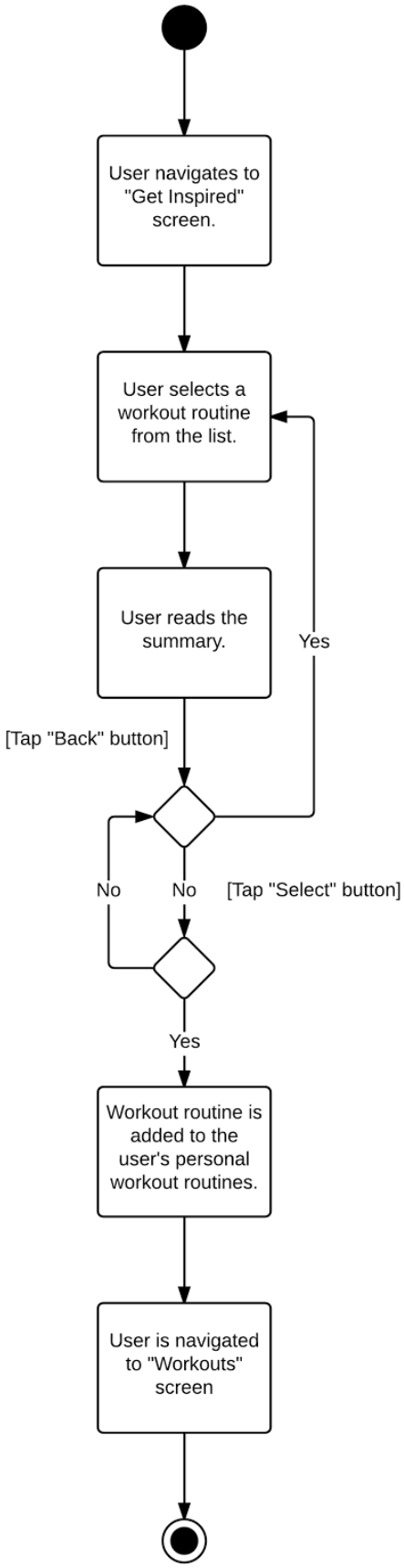
Basic flow

1. User navigates to “Get Inspired” screen
2. User taps on item from the article list
3. User reads the summary
4. User selects Workout Routine to be added to his/hers Workout Routines
5. Application navigates user to Workout List screen

Alternative flow

Condition: returns back

1. User presses “back” button
2. User is returned to article list

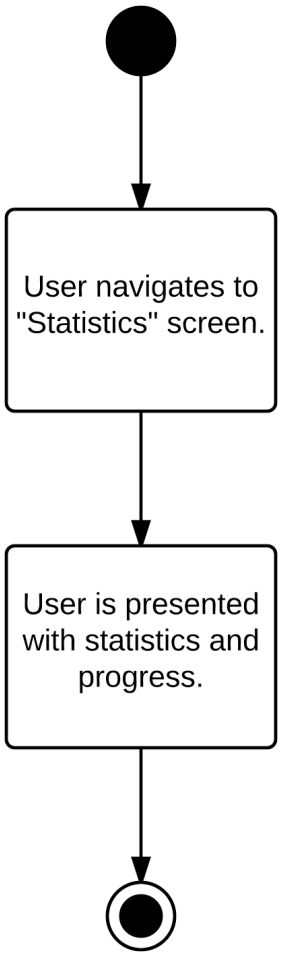

User navigates to "Get Inspired" screen.
User selects a workout routine from the list.
User reads the summary.
Yes
[Tap "Back" button]
No
No
[Tap "Select" button]
Yes
Workout routine is added to the user's personal workout routines.
User is navigated to "Workouts" screen

See Statistics

Use Case Name	See statistics
Use Case Purpose	The purpose of this use case is to allow application user to see his workout progress and statistics
Preconditions	User has launched the application. User has logged in the application.
Postconditions	
Limitations	Progress and Statistics are not shown if none exist
Assumptions	User has iPhone. User has downloaded and installed the application on his phone. User has Wi-Fi or 3G/4G network connection

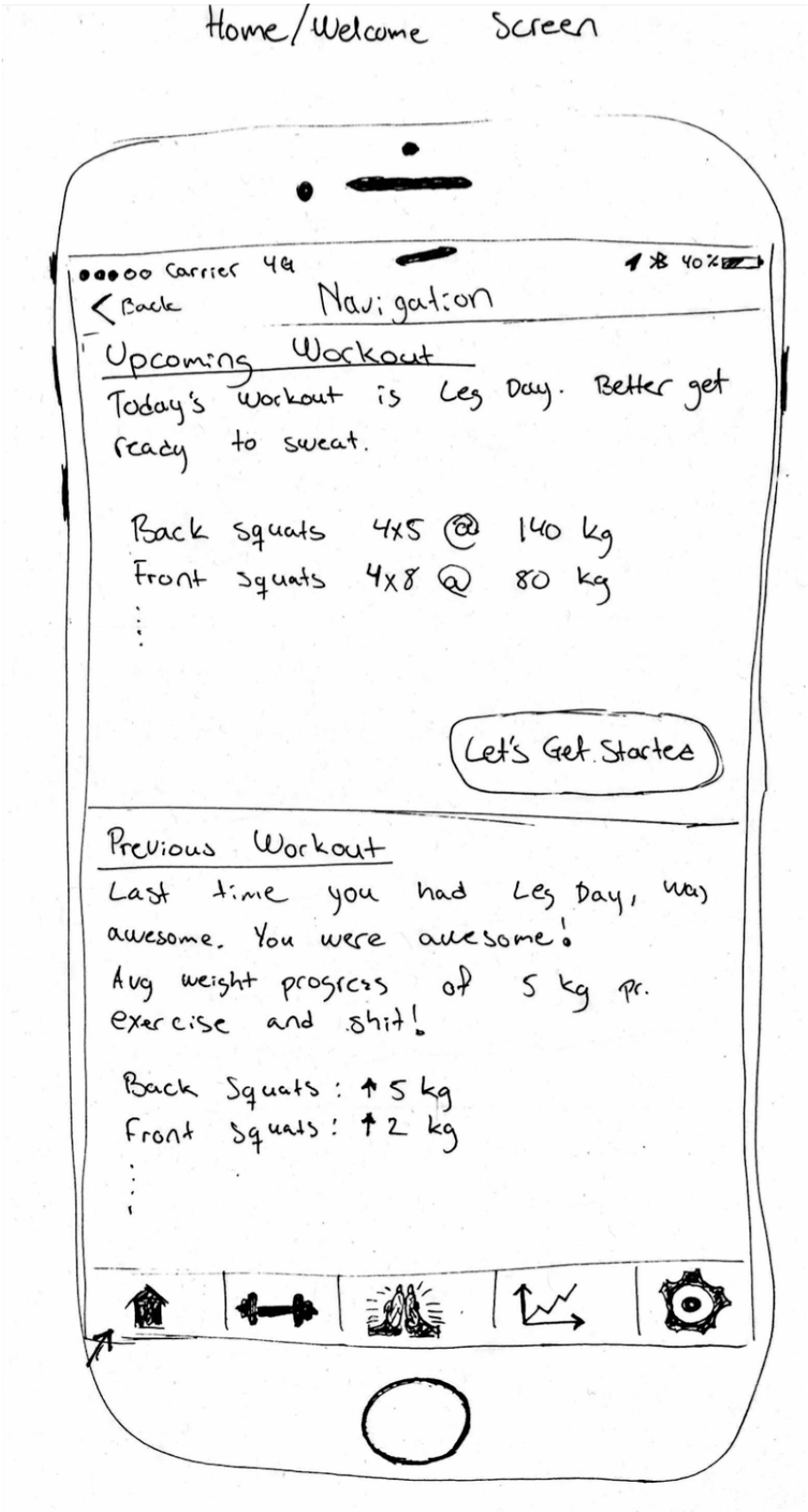
Basic flow

1. User navigates to “Statistics” screen
2. User is presented with workout Summary and statistics

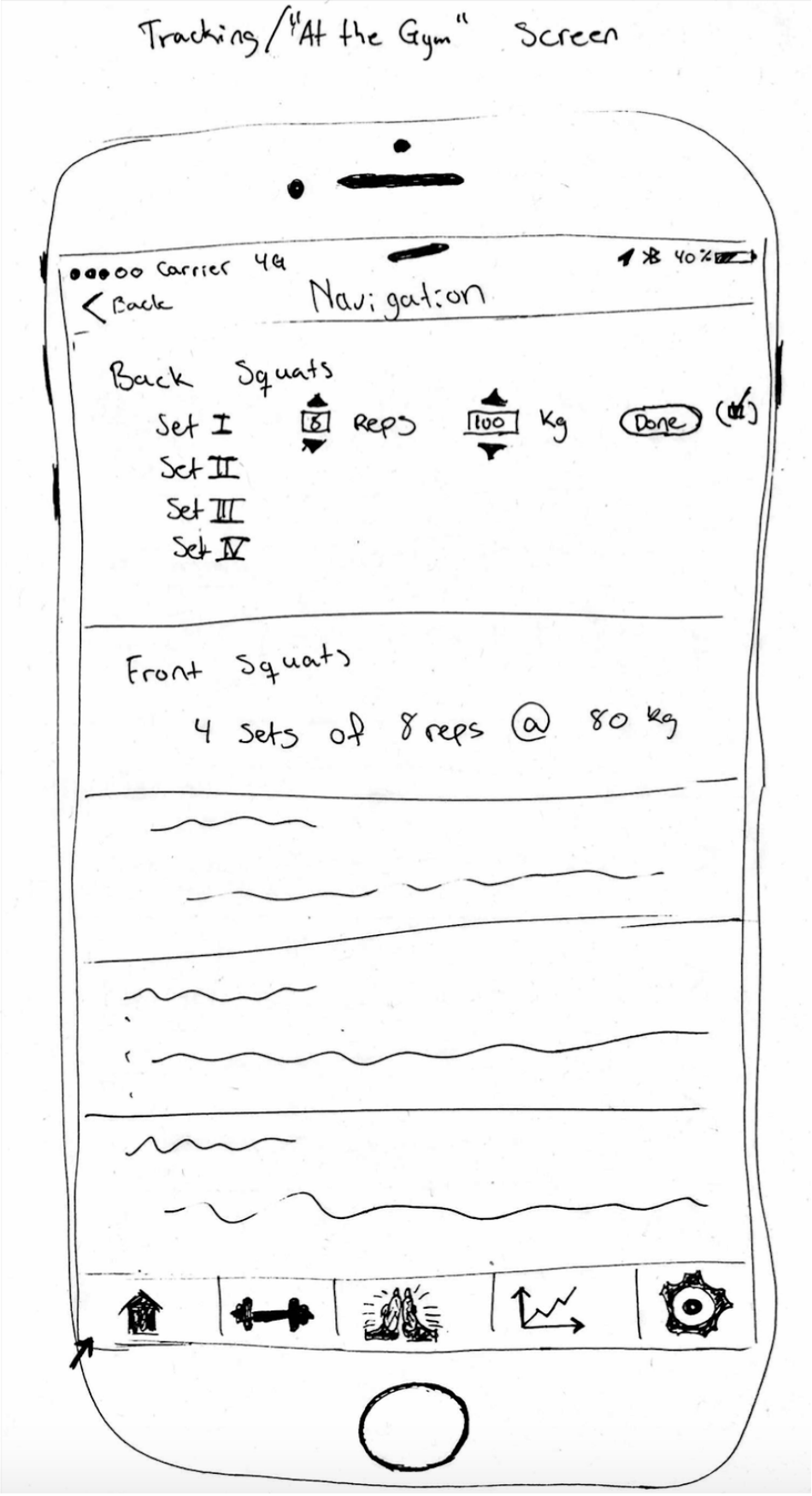


9.3 Appendix C – Graphical User Interface Mockups

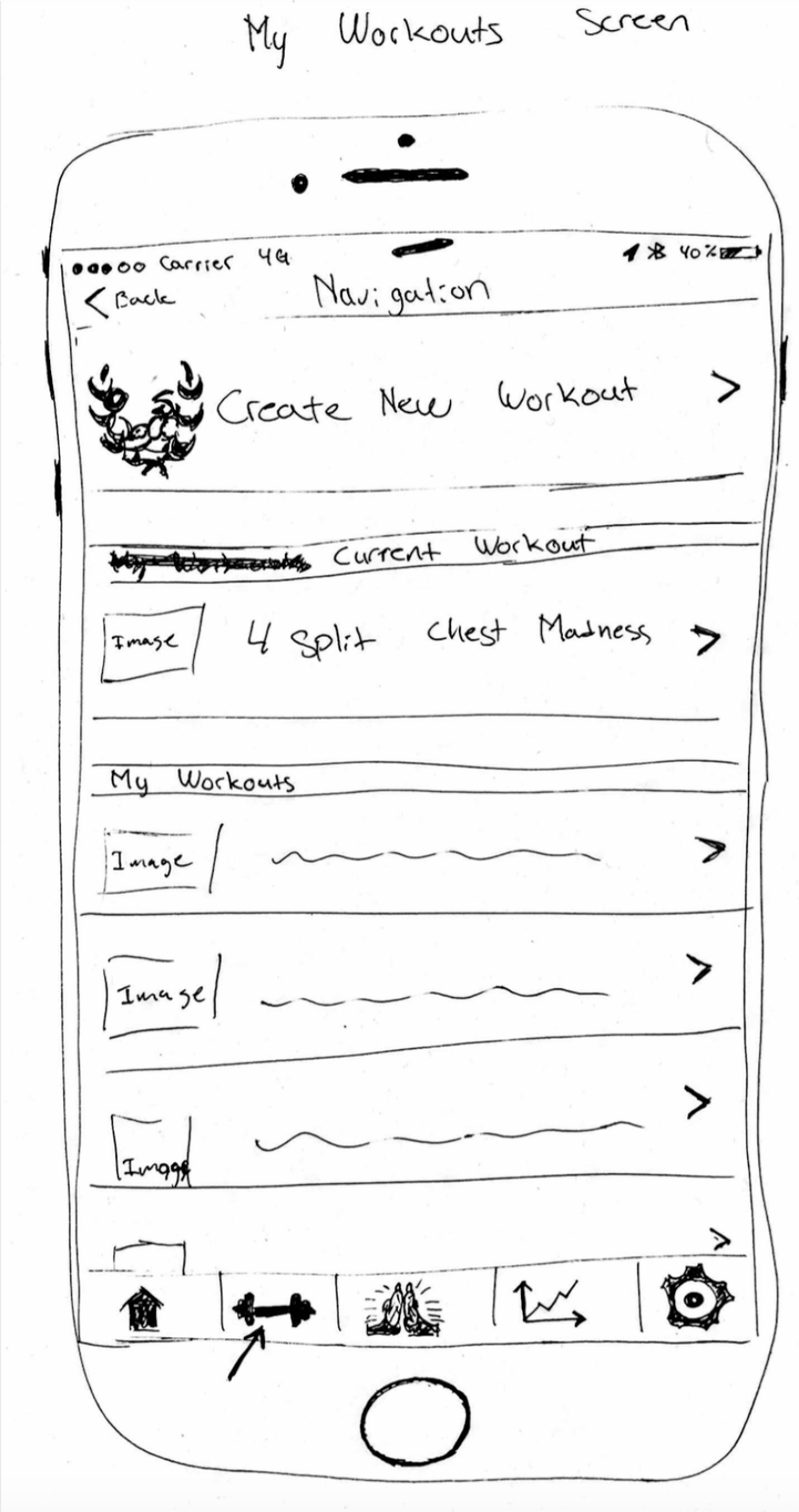
Home Screen



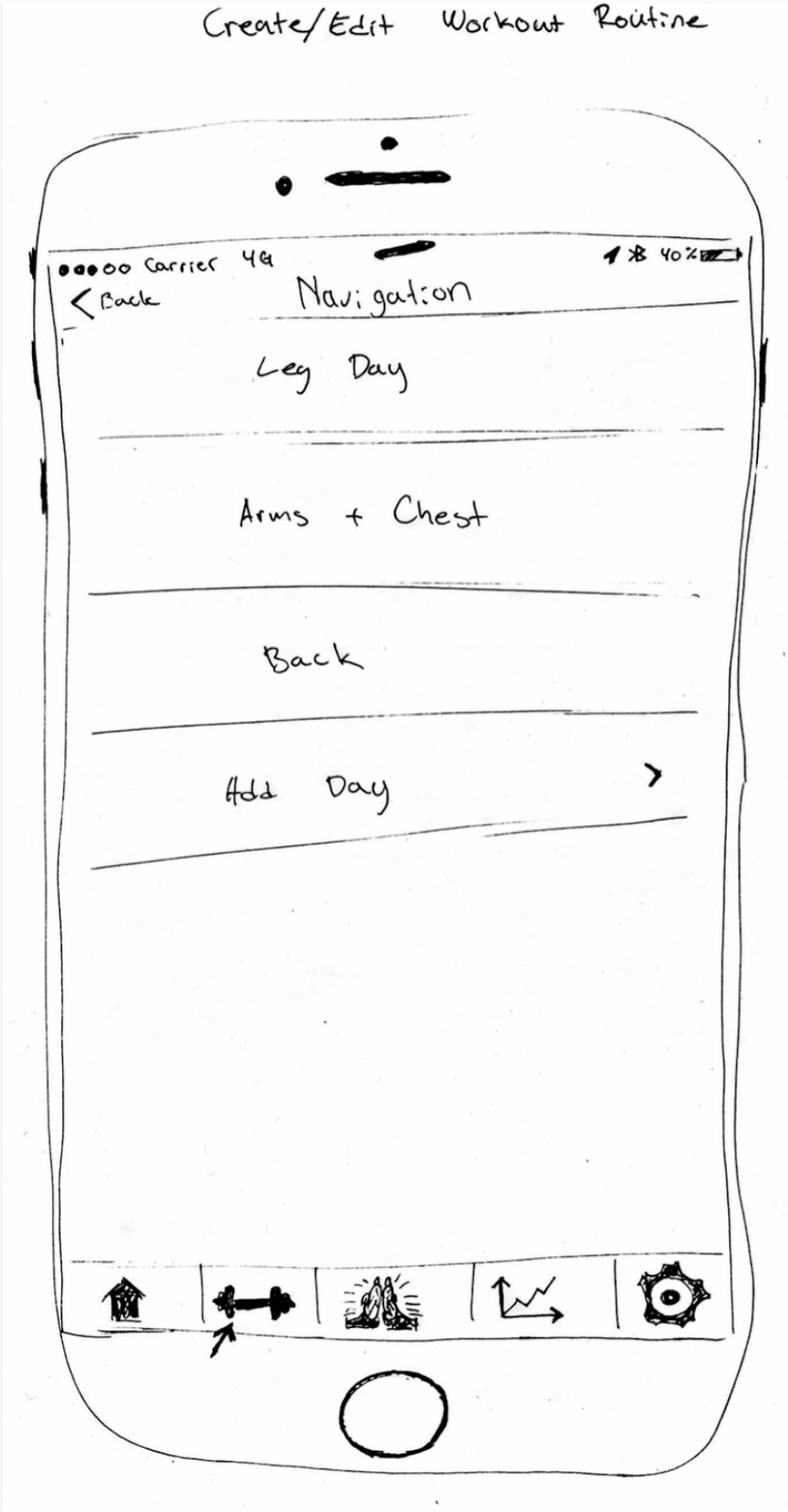
Tracking Screen



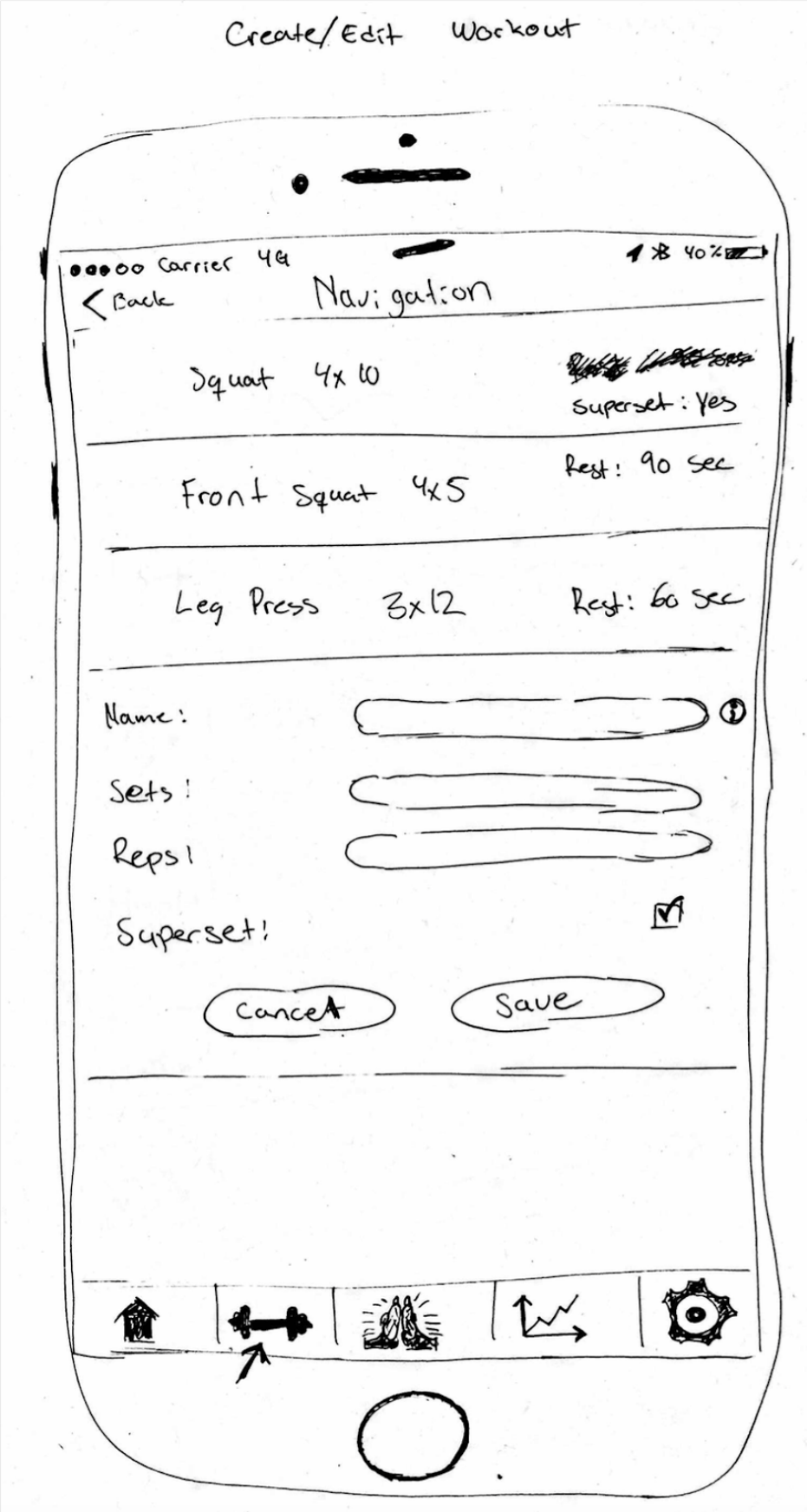
My Workouts Screen



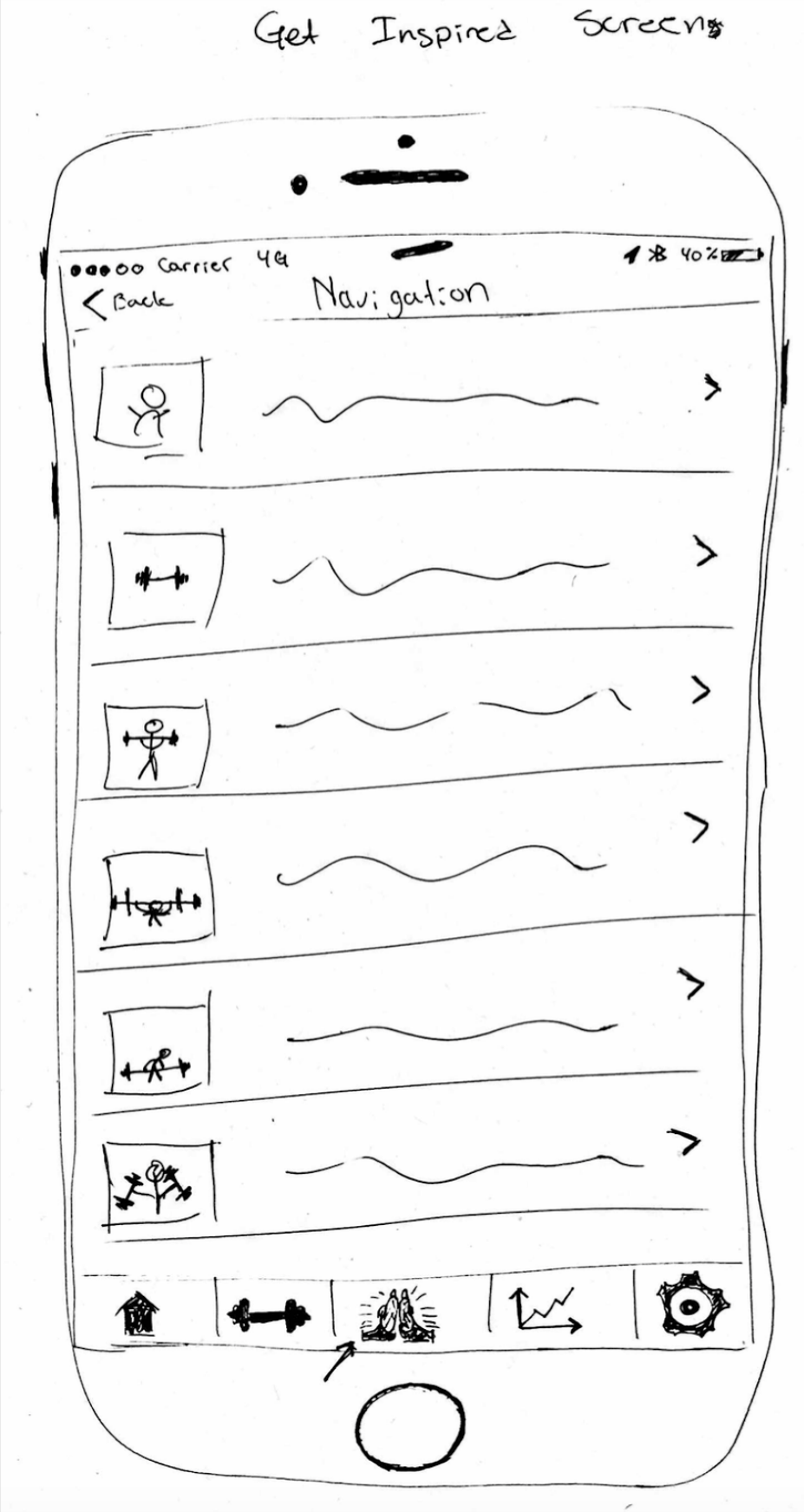
Create/Edit Workout Routine



Create/Edit Workout



Get Inspired Screen



9.4 Appendix D – Xcode Storyboard Overview

This appendix can be found in the Basic Files folder.

9.5 Appendix E – Source Code

This appendix can be found in the Basic Files folder.

9.6 Appendix F – Project Description

This appendix can be found in the Appendix folder.